

IBM CICS Asynchronous API

Concurrent Processing Made Simple

Pradeep Gohil

Julian Horn

Jenny He

Anthony Papageorgiou

Chris Poole





International Technical Support Organization

**IBM CICS Asynchronous API: Concurrent Processing
Made Simple**

December 2017

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (December 2017)

This edition applies to CICS Transaction Server for z/OS Version 5, Release 4.

© Copyright International Business Machines Corporation 2017. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
Authors	ix
Now you can become a published author, too!	x
Comments welcome	xi
Stay connected to IBM Redbooks	xi
Chapter 1. Introduction	1
1.1 Asynchronous processing, parallelism, and concurrency	3
1.2 Why is concurrency desirable?	4
1.3 Models of concurrency	5
1.3.1 Shared state models	5
1.3.2 The actor model, and communicating sequential processes	7
1.4 How does asynchronous processing apply to CICS?	8
1.5 Comparing asynchronous processing techniques in CICS	9
1.6 Summary	10
Chapter 2. The CICS asynchronous API	11
2.1 Basics of the CICS asynchronous API	12
2.1.1 Execute work asynchronously	12
2.1.2 Track the completion of the asynchronously executing work	12
2.1.3 Pass data between parent and child tasks	13
2.2 Four CICS asynchronous API commands	13
2.2.1 The RUN TRANSID command	14
2.2.2 The FETCH CHILD command	14
2.2.3 The FETCH ANY command	15
2.2.4 The FREE CHILD command	16
2.3 Key features and characteristics	17
2.3.1 Transactionality	17
2.3.2 Orphaned child tasks	18
2.3.3 Local children	18
2.3.4 Security model	19
2.3.5 Passing data with CICS channel and containers	19
2.3.6 CICS Asynchronous Services domain	22
2.3.7 Timeouts	23
2.4 Considerations for using the CICS asynchronous API	24
2.4.1 Child GETs and parents UPDATE	24
2.4.2 Allow the same parent program to run and fetch child tasks	24
2.4.3 Long-running parents should use the FREE CHILD command	24
2.4.4 Keep track of fetched channels	25
2.4.5 Review MAXTASK and set transaction classes	25
2.4.6 Parameterizing timeouts	25
Chapter 3. Extending applications while minimizing the impact to response time ..	27
3.1 Overview of the scenario	28
3.1.1 Description of the sample application	28
3.1.2 Objective of the scenario	31

3.2	Add a new request using the CICS asynchronous API	32
3.2.1	Defining the PTNR transaction to run ACCTPTNR	32
3.2.2	Adding logic to print the partner account details	33
3.2.3	Adding the RUN TRANSID command to WEBHOME.cbl	33
3.2.4	Adding the FETCH CHILD command to the WEBHOME.cbl program	35
3.3	Run the updated application	39
3.4	Summary	39
Chapter 4.	Improving the response time of existing applications	41
4.1	Overview of the scenario	42
4.1.1	Description of the sample application	42
4.1.2	Objective of the scenario	44
4.2	Converting program LINKs to asynchronous API calls	44
4.2.1	Define transactions to run the GETNAME and ACCTCURR programs	45
4.2.2	Add RUN TRANSID commands to WEBHOME.cbl	45
4.2.3	Add the FETCH ANY command to WEBHOME.cbl	48
4.3	Running the updated application	53
4.4	Summary	54
Chapter 5.	Developing robust applications with unreliable service providers	55
5.1	Overview of the scenario	56
5.1.1	Objective of the scenario	56
5.2	Requesting services from an unreliable service provider	57
5.2.1	Why not just use a LINK?	58
5.2.2	Asynchronously requesting a new service	59
5.2.3	Testing the response times of calling the new service	62
5.2.4	Retrieving a timeout value to meet the application's SLA	64
5.2.5	Adding the TIMEOUT parameter to the FETCH command of the unreliable service	65
5.3	Running the updated application	69
5.4	Summary	71
Chapter 6.	Creating a Java-based controller in a mixed-language environment	73
6.1	Making promises about the future	74
6.2	CICS asynchronous API classes and methods	76
6.2.1	A golden-path scenario	76
6.2.2	Additional methods: getAny() and freeChild()	78
6.3	Providing a web front end for the web banking application	80
6.3.1	Project setup	80
6.3.2	Program architecture	81
6.3.3	Writing the main program logic	82
6.3.4	Displaying the account details and loan rate	91
6.4	Summary	93
Chapter 7.	Tips and tricks	95
7.1	Trick: Reduce the management burden by running children under a single transaction ID	96
7.1.1	The PARENT program running two different children under the ASCH child transaction ID	97
7.1.2	Using the ASYNCWP wrapper program to extract the target child program from a channel and linking to it	102
7.1.3	The CHILD1 and CHILD2 child programs running under the ASCH transaction	103
7.2	Tip: Run existing COMMAREA-based assets	

asynchronously without changing them	106
7.2.1 The PARENT program running two different children under child transaction ID ASCH passing COMMAREAs to each one.	107
7.2.2 Using the ASYNCWP wrapper program to extract the PROGRAM target child from channel and linking to it with REQUEST-COMM COMMAREA	112
7.2.3 The CHILD1 and CHILD2 child programs running under the ASCH transaction	114
7.3 Tip: Release storage wisely in long-running parent transactions	116
7.4 Trick: Prevent sets of children from interfering in FETCH ANY logic by using FREE CHILD	117
7.5 Tip: Check the status of a child without blocking the parent by using the NOSUSPEND option	118
7.6 Trick: Process as many children as possible in a fixed time period	119
7.7 Tip: Using response-only channels between parent and child transactions	121
Chapter 8. Debugging and problem determination	125
8.1 Using the CICS execution diagnostic facility: CEDF and CEDX.	126
8.2 Asynchronous API abend code.	133
8.3 Tracing asynchronous API applications	133
8.4 Sample application trace flow using FETCH ANY commands	134
8.4.1 The environment.	134
8.4.2 Trace of the PARENT program creating two children.	137
8.4.3 Trace of one child	139
8.4.4 Trace of the PARENT program fetch the response from any child.	140
8.5 Sample application trace flow using FETCH CHILD commands and the NOSUSPEND and TIMEOUT options.	141
8.5.1 The environment.	141
8.5.2 Trace of FETCH CHILD NO SUSPEND	143
8.5.3 Trace of FETCH CHILD TIMEOUT.	143
8.5.4 Trace of FETCH CHILD	143
8.6 Sample application trace flow using FREE CHILD commands.	144
8.6.1 The environment.	145
8.6.2 Trace of free child tasks	145
8.7 Transaction dumps and the asynchronous API	146
8.7.1 Asynchronous parent task transaction dump extract	146
8.7.2 Asynchronous child task transaction dump extract.	147
8.8 System dumps and the asynchronous API	148
8.8.1 Asynchronous parent system dump extract	148
Chapter 9. Performance and management for asynchronous API applications	151
9.1 Special aspects for asynchronous API applications	152
9.2 Managing the number of tasks in the system	152
9.2.1 Using MXT	152
9.2.2 Using TRANCLASS to manage parent transactions.	153
9.2.3 Using TRANCLASS to manage child transactions	153
9.3 Duration of parent tasks in the system	153
9.3.1 Parent tasks waiting upon child tasks	154
9.3.2 MAXTASK condition causing parent tasks to suspend.	154
9.4 Policing parent tasks with CICS policy	155
9.5 Threadsafe considerations	155
9.6 Asynchronous services statistics	156
9.7 Asynchronous services monitoring	156
Chapter 10. System tracking of asynchronous applications	159
10.1 Data gathered by transaction tracking	160

10.1.1	Origin data	160
10.1.2	Previous transaction data	161
10.1.3	Previous hop data	161
10.1.4	Task context data	161
10.1.5	Application context data	161
10.1.6	Flow of tracking data	162
10.2	Using the INQUIRE ASSOCIATION command to track tasks.	162
10.2.1	Building the picture of the application flow using the tracking data.	164
10.3	Using CICS Explorer to track tasks.	164
10.3.1	Tracking interrelated tasks using search.	165
10.3.2	Finding out associated tasks using the Task Associations views	166
10.3.3	Graphical view of associated tasks	169
10.3.4	Graphical view of orphaned tasks	170
10.4	Using CICS Performance Analyzer to understand task relationship.	171
10.4.1	Brief overview of CICS Performance Analyzer	171
10.4.2	Extending the business application.	172
10.4.3	Transaction tracking reports by CICS Performance Analyzer.	172
10.4.4	Transaction group reports by CICS Performance Analyzer	174
10.5	Using IBM OMEGAMON for CICS on z/OS V5.5.0 to monitor performance.	175
10.5.1	Alerts showing up in OMEGAMON	176
10.5.2	Drill down to the problematic task	177

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.


Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

CICS®
CICS Explorer®
DataPower®
DB2®
IBM®

IBM API Connect™
IMS™
MVST™
OMEGAMON®
RACF®

Redbooks®
Redbooks (logo) ®
WebSphere®
z/OS®

The following terms are trademarks of other companies:

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication covers the background and implementation of the IBM CICS® asynchronous API, which is a simple, accessible API that is designed to enable CICS application developers to create efficient asynchronous programs in all CICS-supported languages. Using the API, application developers can eliminate the overhead that is involved in coding and managing homegrown asynchronous solutions, instead using a set of CICS-supported API commands to underpin CICS applications, which are more responsive and robust than ever.

Initially, the book reviews the history and motivations of asynchronous processing in computing and the benefits involved when calling external services. It then introduces the asynchronous API itself and its commands. It also provides a range of scenarios, including sample code, that cover everything from the basics of making an asynchronous request to updating existing synchronous program calls, with the goal of illustrating how to harness the CICS asynchronous API to solve real business problems. Later chapters take a deeper dive into the capabilities of the asynchronous API for advanced use cases.

Beyond application development, CICS provides a complete solution for system programmers to manage and monitor asynchronous business logic. Thus, the final chapters of this book cover enhancements to CICS monitoring, statistics, trace, and dumps. Using supporting CICS tooling, system programmers have greater insight than ever, with improved transaction tracking capabilities and CICS policies to provide maximum control and optimization of asynchronous processing in CICS environments.

Authors

This book was produced by a team of specialists from around the world.

Pradeep Gohil is a Software Engineer working at IBM Hursley UK. He holds a Master of Engineering (MEng) degree in Computing (Artificial Intelligence) from Imperial College London. He has 15 years of experience working in the IBM CICS organization. His area of expertise is application modernization in CICS Transaction Server for IBM z/OS®, including the CICS asynchronous API, web services, CICS cloud, and CICS bundles. Pradeep was the technical leader for the research, design, and delivery of the CICS asynchronous API.

Julian Horn is a CICS software developer working at IBM Hursley UK. He has over 28 years of experience, working on CICS in both the service role and, most recently, as a developer in the CICS asynchronous API team. His expertise spans many of the core areas of CICS including dispatcher, transaction manager, program manager, and security.

Jenny He is a CICS Software Engineer working at IBM Hursley Lab in UK. She started developing the CICS product in 2011. Her areas of expertise includes event processing, CICS policy, CICS asynchronous APIs, IPIC, web service, statistics, and monitoring. She holds a PhD degree in optical communications from University of Essex in UK. She has authored three Redbooks publications prior to this and published a number of papers during her PhD research.

Anthony Papageorgiou is a Software Engineer working at IBM Hursley UK. Before he joined IBM in 2007, he worked for Mintel International Group Ltd as a Web Developer. Anthony holds a degree in Computer Science from the University of Warwick. His areas of expertise include

APIs, mobile, cloud, and event processing technologies. During his time at IBM, he has performed a number of roles across development, architecture, strategy, and management in the CICS organization. Anthony played a key role in the initial research that led to the development of the CICS asynchronous API and is currently working as the delivery manager for z/OS Connect Enterprise Edition.

Chris Poole is an IBM Master Inventor, working on IBM Blockchain, and contributing to the open source Hyperledger project. Prior to this, he worked in the CICS development team, working in the CICS asynchronous API, DevOps, and cloud teams. He continues to develop and evangelize CICS microservices and DevOps technologies. Prior to joining the CICS team, he worked in the IBM API Connect™ development team and developed software to administer IBM DataPower® appliances. He holds a PhD in Theoretical Physics.

This book was managed and edited by:

- ▶ Martin Keen
- ▶ Debbie Willmschen

Thanks to the following people for their contributions to this project:

- ▶ Steve Bolton
- ▶ Amy Reeve
- ▶ Sophie Green
- ▶ Satish Tanna
- ▶ Christopher Walker
- ▶ Geoff Bunworth
- ▶ Ian Burnett
- ▶ Mark Cocker
- ▶ Ivan Hargreaves
- ▶ Paul Cooper

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- Send your comments in an email to:

redbooks@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



Introduction

The CICS asynchronous application programming interface (API) provides a powerful means for IBM customers to optimize CICS applications for Web Services and APIs that are external to CICS. The asynchronous APIs allow for multiple requests to service providers to occur concurrently, rather than one after the other, all using a supported API in CICS.

Previously, CICS applications were single-purpose, self-contained units. These applications typically executed in solitude, and with the throughput and reliability of the platform, the CICS application did its job well—and still does—but technology moves forward. Thus, expectations of the CICS application have grown. Customers want more insight, more capacity, and smarter processing to maintain a competitive edge. Applications that started by taking input from a terminal must now interface with data stores and programs across multiple systems. In addition, they must invoke requests to external services.

Perhaps you started with one simple database of customer details, but over many application enhancements, system consolidations, and business acquisitions spanning several geographies, your application now fetches data across all parts of the organization to build a complete view of the customer. Or maybe a new business objective means that you need to pull together several dispersed services that are provided across the organization's systems.

Organizations today are asking more from CICS applications and want the results sooner. New applications have aggressive response time goals to meet service level agreements (SLAs) and to improve client satisfaction, while existing apps still need to be enhanced without affecting overall response times.

By no means is asynchronous processing a new concept in computing; however, it is still challenging to produce enterprise-grade applications with asynchronous calling patterns. The premise is simple. Instead of calling multiple services sequentially, as illustrated in Figure 1-1, you can reduce the overall response time of the application by requesting services concurrently, as illustrated in Figure 1-2. This process is achieved by running *child work tasks* (that manage a single service request) asynchronously to the parent task.

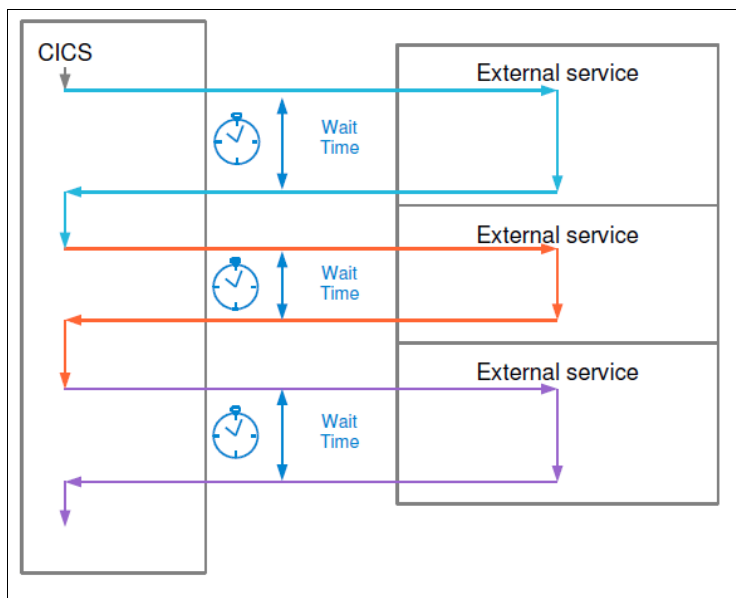


Figure 1-1 Sequential calling pattern of external services

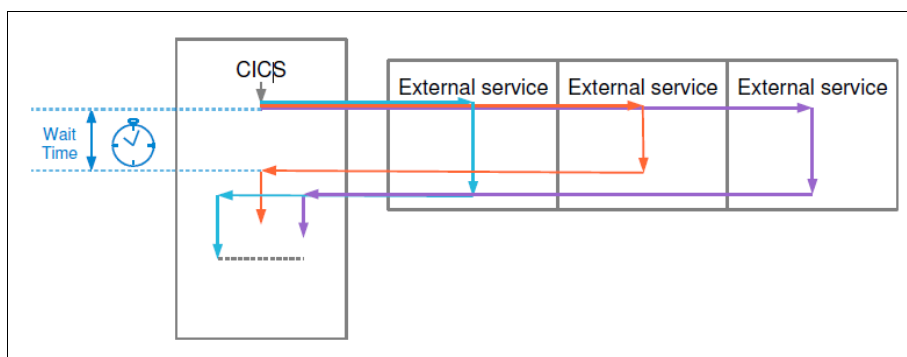


Figure 1-2 Concurrent calling pattern of external services

We've heard numerous experiences of customers who have attempted asynchronous calling patterns in CICS applications, with varying levels of success. We've heard the difficulties involved in coordinating many asynchronous tasks, along with challenges of passing data, problems arising from work becoming out of sync, and the on-going maintenance of home-grown architectures, which is common in high workload, high throughput environments such as CICS.

The CICS asynchronous API enables application developers to deliver asynchronous calling patterns to produce responsive applications that minimize response times. CICS Transaction Server V5.4 introduces the CICS asynchronous API, along with support to control, manage, and monitor asynchronous processing in CICS.

1.1 Asynchronous processing, parallelism, and concurrency

Although this book discusses asynchronous processing, it also provides information about the following related terms:

- ▶ Concurrency
- ▶ Parallelism

It is worth discussing the differences between these terms.

First, compare the following existing CICS commands:

- ▶ **EXEC CICS LINK**
- ▶ **EXEC CICS START**

The **EXEC CICS LINK** command is a *synchronous* command because it runs a named program and gives immediate and predictable control to that program. The calling program has its control returned only at the end of the named program. This processing all runs within the framework of a CICS task.

The term *synchronous* works for this process because if you imagine the calling program and the program being called as each having a clock, it is entirely possible to *synchronize* these clocks. At the point where the “clock” owned by the calling program is paused (when it gives up control and the **EXEC CICS LINK** command is issued), the “clock” owned by the program being called is then started. The same process is true for when the called program stops; its clock stops, and the first clock starts again. They are synchronized.

You can contrast this process to the **EXEC CICS START** command, which names a CICS transaction to start. This process causes an entirely new CICS task to be created (ultimately running a program), without the calling program passing control to it. Thus, if a program within each of these two tasks were to own a clock, you can no longer synchronize the clocks, because you don’t know for certain when the second task will start (while the calling program’s clock keeps ticking). In addition, you certainly can’t predict the time recorded by the first clock of when the second program stops. Because the **EXEC CICS START** command doesn’t pass control from the first program to a second, the command is also said to be *non-blocking*. A program making use of the **EXEC CICS START** command (invoking a local transaction) issues that command and immediately moves to the next line, continuing its operation. (For now, ignore the minor detail that part of the **EXEC CICS START** command is actually being synchronous.)

Now, how does this information relate to concurrency and parallelism? Let’s start with *parallelism*, which is doing lots of things at the same time. There are several caveats that can be applied to this type of processing, because it depends on the architecture and hardware but fundamentally you need several CPUs running at the same time to be able to achieve parallel processing. Of course, this type of processing is the important work of the mainframe, with the z/OS dispatcher and the CICS dispatcher working hard to get this type of work correct. If a program’s logic necessitates that several parts of it *must* run at the same time, it’s a *parallel program*. If you can in theory run this same program on only one CPU, having a preemptive multitasking system to pause tasks and ensure that all logic is run, the program is actually written to be *concurrent*.¹

Concurrency is about dealing with lots of things at the same time. With a truly parallel program and system, things *must* run at the same time. With a concurrent system, they *can* run at the same time, but parts of the computation can advance without waiting for the others to complete. CICS allows for concurrent programs, with support for running code on different

¹ Hardware-level parallelism, such as single-instruction, multiple-data (SIMD), is beyond the scope of this book.

task control blocks (TCBs), such that all processing isn't pipelined onto the quasi-reentrant (QR) TCB. The desired goal of the CICS Asynchronous API is to enable concurrency, where parts of an overall program can run at the same time and reduce overall run time, but to do so with a programming model that's easy to understand.

1.2 Why is concurrency desirable?

There are several reasons why you might want to improve the concurrency of your programs. Imagine that you have an existing program that consists of two logically separate blocks of code. There's no reason why these two separate blocks of code couldn't, in theory, be run at the same time, because the second block is not dependant on the first. If you can run these two blocks of code at the same time, you can reduce the overall response time of the program, as illustrated in Figure 1-3. On the left of the figure is a CICS task with a single program, represented by a dashed line. It is started by something else, perhaps passing data to it, and ultimately returns new data. In the middle of the figure is the same CICS task, where two distinct parts of the program have been identified. Finally, the right side of the figure shows that the two distinct parts of the program, if run concurrently, can result in a reduced response time.

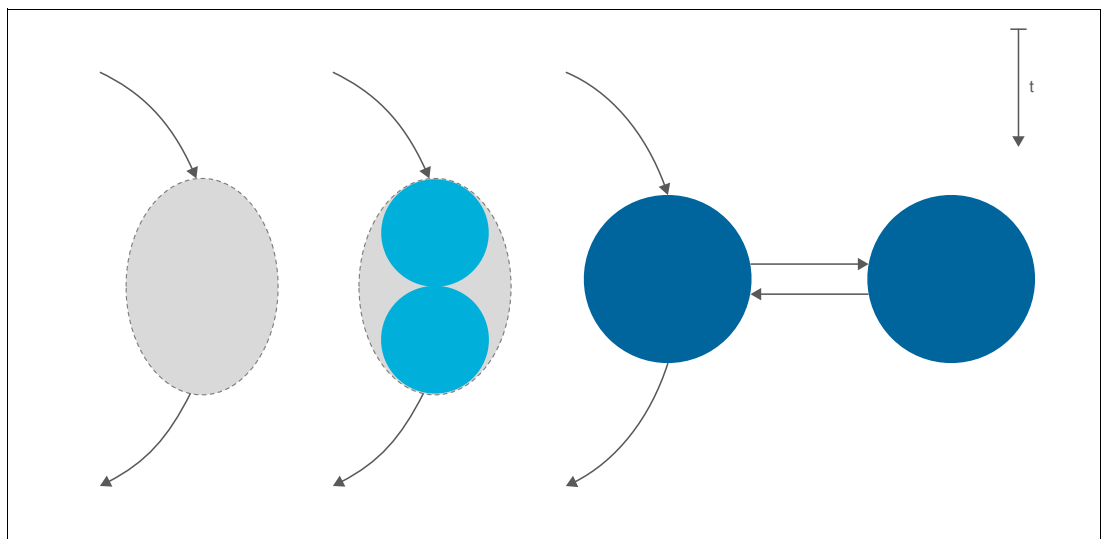


Figure 1-3 A CICS task, with time running from top to bottom

As a second example, consider the fact that CICS is no longer the “end of the chain.” Although CICS functions as the end step of a series of systems that are called, it is also used to call to other services as part of business logic. These other services might be external to the mainframe that CICS is running on and might not 100% reliable. Let's say that you need to make a call to a credit checking agency, but instead of depending on just one agency, you have an agreement to use three agencies. If one call fails, you can still try the others to better tolerate the failure. It would be nice to run these three calls concurrently and get the result of the service that's fastest to respond. This way, you've minimized the response time but also added fault tolerance into the process, as illustrated in Figure 1-4 on page 5. After the same data is sent to each external service, the process requests for the data from the first response back, regardless of which service it happens to come from.

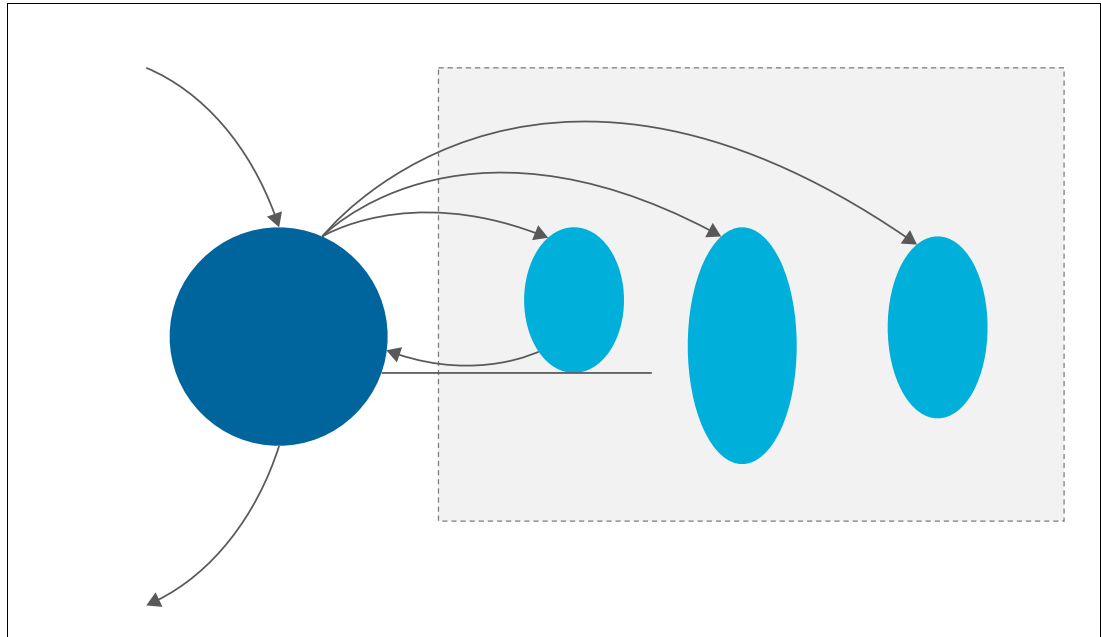


Figure 1-4 A CICS program calling to multiple external services

Finally, consider the business advantages of being able to do things concurrently. Let's take the example of a bank's recent transactions web page. While you're fetching the recent transactions for a customer, perhaps you'd like to see whether they're eligible for a new credit card. This kind of upsell might result in increased business revenue. You can't wait forever of course—you still have to return the customer's recent transactions in a timely manner—so it's desirable to run that processing on the side. But if you can gather this additional information at the same time, why not?

Now that you understand better why concurrency is desirable, the next section discusses the programming models that allow you to achieve it.

1.3 Models of concurrency

To achieve concurrent processing, you need to be able to write programs using a *model* or *framework* that allows for concurrency. Several established methods allow for concurrent processing, including some ways that work initially but that can get complicated quickly. The following sections provide a quick look at two models that allow for concurrent work to be done and an understanding of why there are better methods.

1.3.1 Shared state models

With a *shared state model*, you can make use of one large area of storage that different processes can write to (in a loose, not pointing to any particular implementation sense). For example, if you have a system with two operating system threads, thread 1 can write to this shared state, before starting thread 2. Thread 2 can then read this state information and complete whatever processing it requires, before writing back to the same shared state. This type of processing can be simple to program and efficient. However, if you want to extend this processing beyond two threads, it quickly gets complicated. Furthermore, it introduces a whole class of problems, called *shared state problems*. For example, what happens if two threads try to update the same data in the shared state at the same time? One of the threads

will have its data overwritten. To counter this issue, you can introduce *locking*, but now as an application programmer, you have to manage locks, which introduces further complexity. What happens if these locks get forgotten about? You can reach a deadlock situation if thread 1 locks the state temporarily but dies during its update process. The lock is still in place, but now thread 2 can't update the state itself, introducing even further complications, which is illustrated in Figure 1-5.

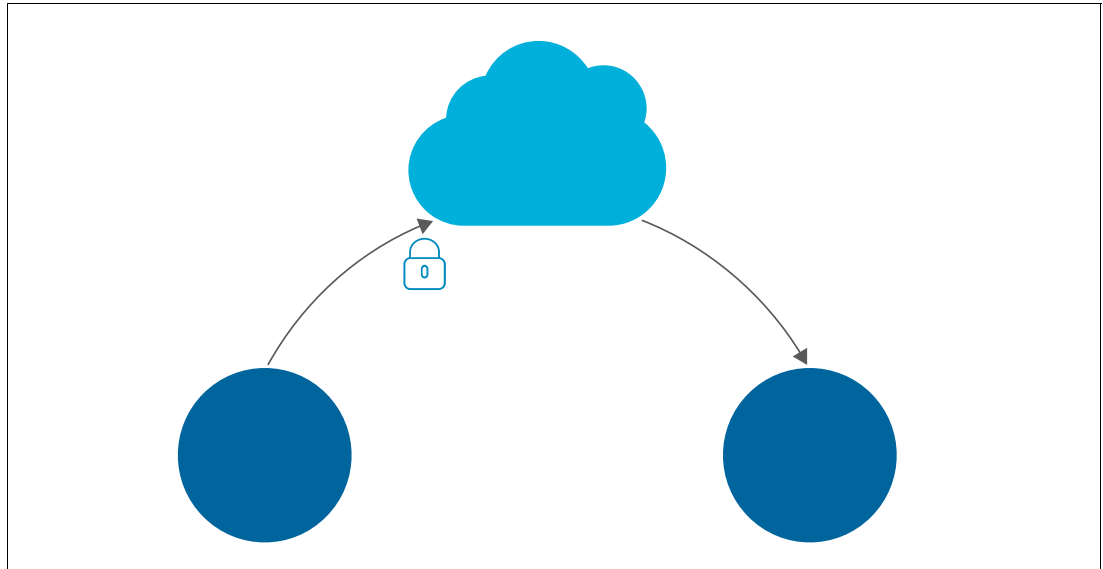


Figure 1-5 A shared state (the cloud shape), to be written to and read by two different processes

Another idea is one of a *shared queue*. Similar to the first example, however, a shared queue can expose you to a subset of the shared state problems. You have a program A create a queue, and then start an asynchronous process of some kind. This asynchronous process, running program B, is about to read information from the queue and push data back onto it. Program A can continue with its processing, while program B is running (concurrently). However, you still have a problem. How does program A know when the data that it needs has been written to the queue by program B? You now need to write polling code that sits and waits for a while before checking the queue and then looping back around, as illustrated in Figure 1-6 on page 7. This method also gives rise to another issue. You write the program to sleep for one second, to check for items on the queue, and then to loop and sleep again. This allows for the possibility that an item will be pushed onto the queue just as the program goes to sleep, leading to poor performance. The program will sleep for a fixed amount of time before it wakes up and is able to read the item from the queue. Ideally, you'd want to have the program read the item from the queue as soon as it is pushed onto it.

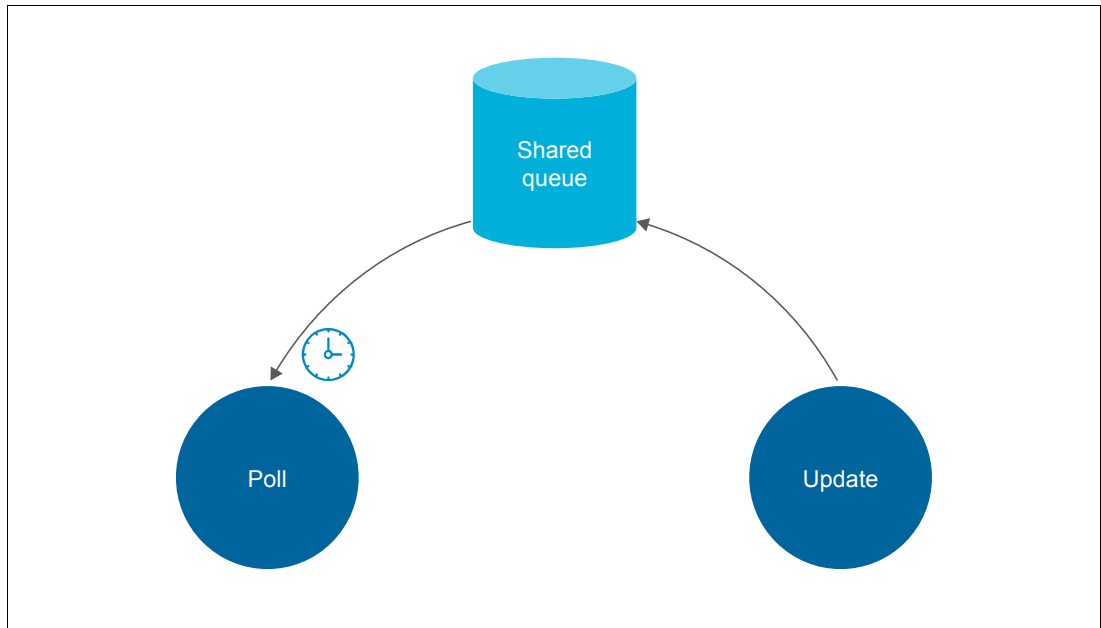


Figure 1-6 A concurrent program, one program writes to and updates a shared queue

Both of these popular solutions to the concurrency problem are susceptible to fundamental issues, which can be caused by the following states:

- ▶ Shared state
- ▶ Mutable state

These problems, among others, have been the subject of much academic research, leading to Communicating Sequential Processes (CSP) and the Actor model. For the CICS asynchronous API, we learned from both, and took preferred practices from both. Let's take a look at the models themselves, then understand why they're a good fit for CICS, and how we've achieved the implementation.

1.3.2 The actor model, and communicating sequential processes

To simplify these models, let's think of a practical example. Dave is an employee at a company and was asked a question in an email from his manager that required an urgent response. Dave received the message in his inbox and now forwards the email with a message to Tony, who he hopes might know the answer to the question. But because he's not sure whether Tony is available or will respond to the email, he also sends the same question to Chris. Dave then continues with his own work, until he has a response in his inbox from either Tony or Chris. (Dave doesn't care who responds to the question; he just wants the fastest response.) When Dave has a response with the answer to his question, he can then send that answer on to his manager. This process makes a lot of sense, which allows us to better understand these models.

An *actor* has the following characteristics:

- ▶ Is something that can do some kind of processing
- ▶ Doesn't share state
- ▶ Communicates explicitly, via message passing semantics

In this example Dave, Tony, and Chris are three actors. Each maintains his own private state, and when they do communicate, they do so by passing messages (emails) to one another, as illustrated in Figure 1-7.

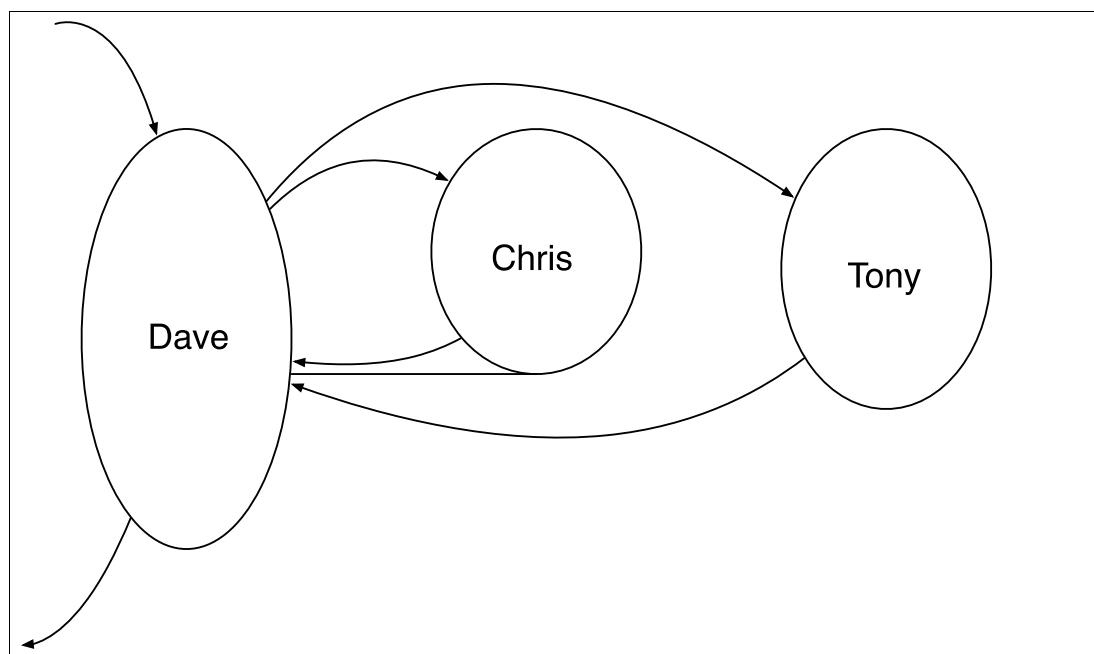


Figure 1-7 Dave sends the same request to Tony and Chris, and then waits for the fastest response

CICS, like the actor model, adopts a buffered mailbox approach. Much like this email example, each CICS task that's started a child task has its own inbox. Beyond CICS, CSP and the actor model have influenced languages and runtimes, such as Google's Golang, Clojure's `core.async`, and Scala's Akka.

1.4 How does asynchronous processing apply to CICS?

This section discusses how to apply the two models of computation discussed previously to CICS. To implement a model with the desired behaviors that are influenced by these models you will need:

- ▶ A run time to handle discrete entities in the system, each able to hold their own private state information (the *actors*, as such)
- ▶ An ability to pass messages between these entities

CICS tasks solve the first requirement for you. These tasks are discrete entities in the system, and the CICS dispatcher already does a good job of managing them. The dispatcher takes care of running these tasks on different TCBs as it usually does, given the correct transaction definitions for thread safety. The aim isn't to rewrite everything; if something works well and is time-tested, use it. This reasoning also explains why a buffered mailbox approach, similar to the actor model, is a good approach. If you have a task (task A) ready to return a message to another task (task B), you don't want task A to sit around until task B is ready to receive this message. These are regular CICS tasks, using up task slots, a system-limited resource. So, allow the message to be delivered, and task A to be cleaned up as usual.

The second requirement is also solved for you already. CICS TS V3.1 introduces channels and containers. You can make use of this same channel technology to pass messages from

one task to another. You can also reach a solution that helps with desired fault tolerance goals, where one actor, if it dies, does not kill any other actor that it is connected to. CICS tasks are equally robust.

To help you achieve this kind of processing in your own programs, we've implemented this processing model using the following API commands:

- ▶ **EXEC CICS RUN TRANSID**
- ▶ **EXEC CICS FETCH CHILD**
- ▶ **EXEC CICS FETCH ANY**
- ▶ **EXEC CICS FREE CHILD**

1.5 Comparing asynchronous processing techniques in CICS

The CICS asynchronous API isn't the only way of carrying out asynchronous tasks. In fact, you've been able to run asynchronous workloads in CICS for many years. This section briefly analyzes some of the other processing techniques in CICS to ultimately understand why the CICS asynchronous API was created.

One pattern that some CICS users employ is that of **EXEC CICS START** command invocations, and listener tasks. Listener tasks can sit and wait for a request to come in to it and then start a child task that is related to that request asynchronously. This process works well, but it is tricky to get information back from these child tasks. Additionally, the **EXEC CICS START** command isn't threadsafe and, for equivalent usage, is less performant than the **EXEC CICS RUN TRANSID** command. (It does offer parameters for starting the task under a different USERID, and at a later time, though.)

To collect results back from child tasks, queues are often the solution that are turned to. A parent task can start a task asynchronously with the **EXEC CICS START** command and have that child task write its output to a shared queue (for example, a temporary storage (TS) queue). The parent task can then listen, or *poll*, for updates to that queue. To some extent this process allows for recoverability, because the queues can be persisted. However, it is not a trivial process. This method adds complexity because there are now new CICS resources that must be managed and maintained by CICS system programmers, in addition to the code to manage those queues. There's also the question of what happens if a child task abends before it pushes to the queue. How can the parent determine this occurrence? You can also use messaging middleware, such as IBM MQ, but using this method suffers from similar issues. In addition, you will have another product to maintain and another skill to learn, if this is its only use in the organization.

If a greenfield project is developed, you can consider native Java APIs (such as with the `java.util.concurrent.Future` interface). If the logic needs to be written only in Java and doesn't interact with programs that are written in other languages within a CICS region, this method can work well (allowing for the management of JVMSERVER resources and so on). However, it becomes less practical to use this process if you also want to make use of existing, reliable business logic, written in COBOL or PL/I perhaps.

Finally, CICS Business Transaction Services (BTS) offers an asynchronous model that can be used here with success. The framework is designed more for long-running, pseudo-conversational programs, however, and programs must be adjusted to be written in this style. Adjusting programs is not as trivial as adding an invocation of the **EXEC CICS RUN TRANSID** command. For an expert user, Event Control Block (ECB) POSTs and WAITs can be employed, but this method requires an understanding of assembler. Although not difficult for a skilled person to write for one child task, this task becomes more challenging for many child tasks.

1.6 Summary

In summary, you can use many methods to construct your own asynchronous framework. This framework typically involves the technologies discussed in this chapter, often outside of their core intended use cases. Furthermore, you need to maintain the infrastructure. For these reasons, the CICS asynchronous API was developed. Starting a child task is as easy as using the **EXEC CICS RUN TRANSID** command. Fetching those results are equally as easy, together with the completion status of a child task, even if the child task abended. And although queues and events are used within the CICS run time to manage this process, this process is not something that has to be maintained by a CICS system programmer or understood by a CICS application programmer.

This book provides information about how the CICS asynchronous API fits together and how to get the most out of it. The next chapter introduces the API, and the next several chapters that follow build an example, ultimately leading to the development of a web app that displays bank account information. The book also includes the considerations for system programmers and the use of supporting tools.



The CICS asynchronous API

Chapter 1, “Introduction” on page 1 described the background of asynchronous programming. By harnessing these asynchronous techniques, you can improve the responsiveness and robustness of CICS applications, especially those applications that suspend execution due to waiting upon service calls, while they could be getting on with other useful work.

Although asynchronous processing is not a new concept, there remain many challenges to writing quality and scalable asynchronous algorithms. CICS Transaction Server V5.4 introduces a set of CICS asynchronous API commands that greatly simplify coding asynchronous applications. These commands provide the ability to:

- ▶ Initiate and reconcile asynchronous work
- ▶ Manage the passing of data
- ▶ Mitigate problems related to timing windows, especially in high-throughput workloads

This chapter delves deeper into the set of asynchronous API commands in CICS. It begins by introducing the CICS asynchronous API and then explores key concepts and features, followed by considerations for its usage. Later chapters show working examples of how to use the API and how to work with asynchronous patterns in CICS.

2.1 Basics of the CICS asynchronous API

The CICS asynchronous API is a set of the following EXEC CICS API, and supporting JCICS, commands:

- ▶ **RUN TRANSID**
- ▶ **FETCH CHILD**
- ▶ **FETCH ANY**
- ▶ **FREE CHILD**

This set of commands enables developers to rapidly create asynchronous processing logic in their CICS applications, across all supported languages.

Along with these commands, a comprehensive set of CICS features to manage asynchronous workloads in CICS includes the following features:

- ▶ Automated CICS control on asynchronous workloads
- ▶ Security context flowing
- ▶ Enhancements to trace and dumps
- ▶ Monitoring fields
- ▶ Statistic fields
- ▶ Control with CICS policy
- ▶ Tracking, visualizations, and reporting via tooling

At the heart of the CICS asynchronous feature is the *parent-child model*. Parent tasks run child tasks to execute logic asynchronously to the parent. The parent can, at a later point in the algorithm, fetch back the results from a completed child task.

Parent transactions have one or more child transactions. Child transactions have a single parent transaction.

Asynchronous processing in CICS include the following abilities:

- ▶ Execute work asynchronously
- ▶ Track the completion of the asynchronously executing work
- ▶ Maintain data integrity across tasks

2.1.1 Execute work asynchronously

A parent task issues the **RUN TRANSID** command to initiate a child task. In this regard, it is similar in nature to a basic **START** command. The difference between a **RUN TRANSID** command and the **START** command, is that the **RUN TRANSID** command provides the parent task with a “handle” to obtain the status of the child task.

It is important to note that following the execution of the **RUN TRANSID** command, the parent task is not blocked awaiting a reply and might continue to execute. For example, the parent might choose to execute other business logic, request results from the child task, or even initiate further child tasks.

2.1.2 Track the completion of the asynchronously executing work

When a parent task requires the completion status or results from a child task, it can issue a **FETCH CHILD** command. As an alternative to requesting a specific child, a parent can issue a **FETCH ANY** command to return the results from any of the child tasks that it has initiated.

A parent can signify a lack of future interest in a child task by issuing a **FREE CHILD** command. The parent task is then not notified regarding freed child tasks on subsequent **FETCH CHILD** or **FETCH ANY** commands. A freed child does not abend nor stop execution because of the **FREE CHILD** command and continue execution independently.

2.1.3 Pass data between parent and child tasks

Data can be passed from a parent to a child (and returned) using a CICS channel and containers. The CICS system manages the channel data between parent and child tasks.

Figure 2-1 shows a parent transaction that initiates three child transactions. The three child transactions run independently to the parent transaction. After issuing three **EXEC CICS RUN TRANSID** commands, the parent still executes business logic until responses are required from the child tasks. At that point, the parent issues an **EXEC CICS FETCH ANY** command and suspends until a child completes. This process is repeated two further times to consume the results of all child tasks. The parent then executes more business logic and finally completes.

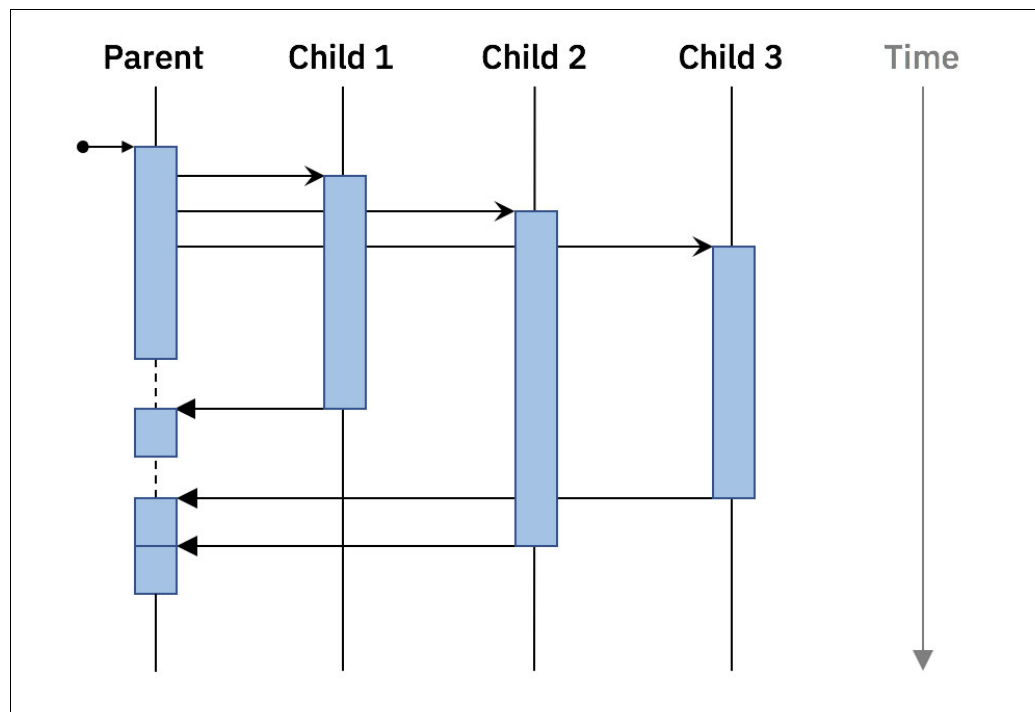


Figure 2-1 A parent task runs three child tasks asynchronously

2.2 Four CICS asynchronous API commands

CICS TS V5.4 introduced the following API commands, known as the *CICS asynchronous API commands*, to greatly ease the development of asynchronous processing patterns in CICS applications:

- ▶ **RUN TRANSID**
- ▶ **FETCH CHILD**
- ▶ **FETCH ANY**
- ▶ **FREE CHILD**

This section provides a closer look at each of the commands in turn and their syntax. For further details about these commands, refer to [IBM Knowledge Center](#).

Along with the four CICS asynchronous API commands, a JCICS equivalent implementation for asynchronous processing in Java has also been provided in CICS TS V5.4. You can find further details about the JCICS implementation in Chapter 6, “Creating a Java-based controller in a mixed-language environment” on page 73.

2.2.1 The RUN TRANSID command

The **RUN TRANSID** command initiates a local child transaction that runs asynchronously with the parent transaction. Example 2-1 shows the syntax.

Example 2-1 Syntax for the RUN TRANSID command

```
>>-RUN--TRANSID(name)--+-----+--CHILD(data-area)-----><
                        '-CHANNEL(name) -'
```

The TRANSID(name) parameter is an input field and must specify the 1 to 4 character transaction identifier for the child task you intend to run asynchronously. This transaction must be a local transaction.

The CHILD(data-area) parameter is an output field and is populated by CICS after issuing the API command. This child token is used by future asynchronous API commands to identify the child instance.

The optional CHANNEL(name) parameter is an input field where you specify which channel is to be made available to the child task. The presence of the CHANNEL parameter indicates that data is passed to or from the child task.

2.2.2 The FETCH CHILD command

The **FETCH CHILD** command is used by a parent task to inquire on the status of a specific child task, and returns the status of the specified child task. Example 2-2 shows the syntax.

Example 2-2 Syntax for the FETCH CHILD command

```
>>-FETCH--CHILD(data-value)--+-----+----->
                                '-CHANNEL(data-area) -'
>--COMPSTATUS(cvda)--+-----+----->
                        '-ABCODE(data-area) -'
>--+-----+-----><
    +-NOSUSPEND-----+
    '-TIMEOUT(data-value) -'
```

The CHILD(data-value) parameter identifies which child the **FETCH CHILD** command is expected to act upon. This input value is the returned child token from the **RUN TRANSID** command.

The required `COMPSTATUS(cvda)` parameter returns the completion status of the completed child task. Do not confuse this parameter with the standard `RESP` and `RESP2` return codes for API commands, which signify the success of the command itself. The child task returns one of the following completion statuses:

- ▶ `NORMAL`
- ▶ `ABEND`
- ▶ `SECERROR` (a security error)

If the child task has abended (signified by the `ABEND` value for the `COMPSTATUS` attribute), the optional `ABCODE(data-area)` output parameter contains the abend code thrown by the child task.

To retrieve data in a channel back from a completed child task, you must specify the optional `CHANNEL(data-area)` parameter. This parameter is an output field and is updated by CICS with the name of the channel that contains the results of the child task. By using the `CHANNEL(data-area)` parameter, the channel is bound to the issuing parent program. You can bind a child's channel back to a parent only once (although, a parent task may have multiple channels bound to it from multiple child tasks). You can then use the returned channel name in subsequent **GET CONTAINER** command and other API commands.

By default, the **FETCH CHILD** command is a blocking command. If the targeted child task is still executing, the parent issuing the **FETCH CHILD** command suspends until the child task completes. A time limit can be placed on the duration the parent task is willing to wait upon a child's completion. Specifying the optional `TIMEOUT(data-value)` input-field, the parent program regains control as soon as the child completes or when the timeout duration has been exceeded. Alternatively, you can specify the `NOSUSPEND` option, which prevents the command from blocking and which can be useful to inquire on child tasks.

Tracking returned channel names: Specifying the `CHANNEL` parameter binds the child's channel to the parent. Because you can bind a child's channel to a parent task only once, you need to ensure that your application's logic tracks any returned channel names. This tracking is particularly pertinent in inquiry scenarios with the use of **NOSUSPEND** command, if the intention is simply to check whether a child has completed.

An alternative approach is to issue multiple **FETCH** commands to inquire and to retrieve the data when required. For example, you first issue the following command:

```
EXEC CICS FETCH CHILD(child1)
      COMPSTATUS(child1-comp-status)
      NOSUSPEND
```

Then, follow later with this command:

```
EXEC CICS FETCH CHILD(child1)
      COMPSTATUS(child1-comp-status)
      CHANNEL(child1-channel)
```

2.2.3 The **FETCH ANY** command

The **FETCH ANY** command is used by a parent task to inquire on the status of any child task, and returns the status of any completed child task that has not yet been fetched. The **FETCH ANY** command allows for the greatest response time savings by enabling parent tasks to consume the results of child tasks as soon as they are available, rather than naming which child to process next. Example 2-3 on page 16 shows the syntax.

Example 2-3 Syntax for the *FETCH ANY* command

```
>>-FETCH--ANY (data-area)--+-----+----->
                                '-CHANNEL (data-area)- '
>--COMPSTATUS (cvda)--+-----+----->
                                '-ABCODE (data-area)- '
>--+-----+----->
    +-NOSUSPEND-----+
    '-TIMEOUT (data-value)- '
```

Unlike the **CHILD**(data-value) parameter on the **FETCH CHILD** command, the **ANY**(data-area) parameter on the **FETCH ANY** command is an output field. The **ANY**(data-area) parameter is updated by CICS to identify which child instance has been provided to the parent. This instance matches one of the **CHILD**(data-area) instances from the **RUN TRANSID** commands.

The required **COMPSTATUS**(cvda) parameter returns the completion status of the completed child task. Do not confuse this parameter with the standard **RESP** and **RESP2** command return codes for API commands, which signify the success of the command itself. The child task returns one of the following completion statuses:

- ▶ **NORMAL**
- ▶ **ABEND**
- ▶ **SECERROR** (a security error)

If the child task has abended (signified by the **COMPSTATUS** parameter of the **ABEND** command), the optional **ABCODE**(data-area) output parameter contains the **ABEND** command code that is thrown by the child task.

To retrieve data in a channel back from a completed child task, you must specify the optional **CHANNEL**(data-area) parameter. This parameter is an output field and is updated by CICS with the name of the channel that contains the results of the child task. By using the **CHANNEL**(data-area) parameter, the channel is bound to the issuing parent program. You can bind a child's channel back to a parent task only once. However, a parent task might have multiple channels bound to it from multiple child tasks. You can then use the returned channel name in subsequent **GET CONTAINER** commands and other API commands.

By default, the **FETCH ANY** command is a blocking command. If all un-fetched child tasks are still executing, the parent issuing the **FETCH ANY** command suspends until a child task completes. A time limit can be placed on the duration that the parent task is willing to wait upon the child task's completion. Specifying the optional **TIMEOUT**(data-value) input-field, the parent program regains control as soon as a child completes or when the timeout duration has been exceeded. Alternatively, you can specify the **NOSUSPEND** option, which prevents the command from blocking and which can be useful to inquire on child tasks.

Tracking returned channel names: A **FETCH ANY** command with the **NOSUSPEND** and **CHANNEL** options specified binds a completed child's channel to the parent. Because you can bind a child's channel to a parent task only once, ensure that your application logic tracks any returned channel names.

2.2.4 The **FREE CHILD** command

The **FREE CHILD** command frees a specified child token that was previously allocated by a **RUN TRANSID** command. If a parent task no longer requires the response of an executing child task, using the **FREE CHILD** command frees the resources that are associated with that child

task when it completes, rather than waiting for them to be fetched. The command does not affect the execution of the child task.

If the parent has previously bound the child's channel to the parent (by use of the **FETCH** command with the **CHANNEL** parameter), it is the responsibility of the parent task to discard the channel if it is no longer required. Otherwise, the channel is discarded by CICS when the parent task terminates.

Example 2-4 shows the syntax.

Example 2-4 Syntax for the FREE CHILD command

```
>-FREE--CHILD(data-value)-----><
```

The single, required **CHILD(data-value)** input parameter specifies the child to be freed. This child field will have been returned on the prior **RUN TRANSID** command.

2.3 Key features and characteristics

This section takes a deeper look at various characteristics of the CICS asynchronous API commands.

2.3.1 Transactionality

At a high level, all parents and children are themselves CICS *transactions*, between which CICS maintains a relationship. Being their own transactions means that parent tasks and child tasks have their own lifecycles, or in CICS' terms, their own *unit of work* (UOW). A transaction is a well-known concept in CICS programming and, thus, provides a solid and predictable basis to parent and child tasks for asynchronous processing in CICS. It is important to appreciate this characteristic because it can shape how the application is constructed.

A parent task issues a **RUN TRANSID** command or commands to run one or many child tasks. These child tasks execute independently to the parent. The child transactions are subject to their own processing and dispatching cycles. Thus, at any point, a parent is not aware if the child is executing, suspended, still to be dispatched, or completed. Similarly, a child task is unaware if the parent is still executing or completed.

A parent can obtain the results of a child using one of the **FETCH** commands. A primary attribute on the **FETCH** command is the **COMPSTATUS** parameter, which indicates the completion status of the child. Furthermore, data can be returned to the parent in a channel only if the child has completed. If a parent task issues a **FETCH** command that results in a **COMPSTATUS** of **NORMAL**, **ABEND**, or **SECERROR**, then the parent can safely ascertain that the child UOW has completed.

A child is free to terminate at any time. It is this feature that allows asynchronous replies to be consumed in a timely manner. For information about how CICS maintains the completed child's channel for the parent's later consumption, refer to 2.3.5, "Passing data with CICS channel and containers" on page 19.

Note: A common misunderstanding is an expectation that when a parent terminates that the remaining child tasks abend, cancel, or somehow roll back. Although this expectation sounds practical in theory, in practice it might have the following unfortunate consequences:

- ▶ To “join” all the UOWs, all child and parent tasks must remain active or suspended in the system (thus retaining locks and resources) until all child and parent tasks are completed. This process would have had the net effect of hogging resources, of not releasing task slots, and of creating unresponsive services that might cause major blockages.
- ▶ Child tasks execute at different rates. Some might not have even started, some might be executing, and others might have already completed and terminated. To cancel only those that are in-flight results in an indeterministic set of child tasks, causing confusion.
- ▶ Take an example of a child that simply issues a web service request. Apart from a small window before and after the service call, for much of the time, this request is suspended and unable to abend. Alternatively, the “rug would be pulled out from under” the web service call if the requester has abended. The resulting bad response, or abend percolation, across the system is likely to be more expensive than simply allowing the child to continue and to ignore the result.

Adopting the current architecture, where each parent and child is its own UOW and the termination of the parent does not affect a child, allows a more flexible, deterministic, and responsive parent-child model.

2.3.2 Orphaned child tasks

As previously discussed, child tasks have their own UOW. In addition, child tasks will not prematurely end if the parent task terminates. Child tasks that continue beyond their parent’s task termination are termed *orphaned child tasks*. Orphaned child tasks might be the result of error scenarios. For example, the child did not respond in a timely manner and was timed out by the parent, or perhaps the parent abended and orphaned all in-flight child tasks. Orphaned child tasks might also be a result of expected high-availability architectures, such as requesting a response from three service providers and only requiring the first to respond.

In most cases, orphaned child tasks do not pose an issue. Their natural task termination is managed by CICS to clear storage and dispose of channel data. Ordinarily, tasks are short lived, and allowing them to end naturally is better for the system than attempting to force them out. However, if a task is stuck and other timeout features cannot intervene, an operator can issue standard termination commands, as per any regular CICS transaction. In these error scenarios, the transaction tracking feature can be of use to identify any related transactions currently in the system.

2.3.3 Local children

Parent transactions run child transactions by issuing the **RUN TRANSID** command. All child transactions are local to the parent. Thus, child transactions are always run in the same CICS region as the parent.

When the child task is executing, it is a full-feature transaction and can go remote by using the regular CICS features, if needed. For example, if you had a parent task in a terminal-owning region (TOR) and intend to run work asynchronously in the Application Owning Region (AOR), issue the **RUN TRANSID** command in the TOR parent to run an asynchronous child task locally in the TOR. The TOR child then executes a distributed program link (DPL) to the AOR

to run the bulk of the workload asynchronously. This process results in two tasks in the TOR and one in the AOR.

Note: We often get asked about the **RUN TRANSID** command behaving as a remote **START** command. For example, the ability for a parent in the TOR to run a child in an AOR (without the need for a local child in the TOR). Simply put, this is not workable in practice.

To maintain the relationship between parent and child tasks (for tracking, future retrieval, management, and so on) a certain amount of state data needs to remain in the local region. Couple this need with information for users to work with asynchronous patterns and by the time this is done, we would have reinvented something akin to transactions. Remote STARTs also pose a challenge with robustness with poorly performing remote systems, and work load management scenarios.

A driving force during the design of the asynchronous API was to keep it simple. Simple to understand and to use. This need resulted in the easy to comprehend “all child tasks are local and are able to DPL remote if needed.”

2.3.4 Security model

Parent transactions run child transactions by issuing the **RUN TRANSID** command. With resource level security active, a parent transaction can be configured to call an external security monitor (such as IBM RACF®) to check the parent is authorized to run the child transaction. The child transaction runs under the same user ID as the parent transaction.

You can find more information about security and the **RUN TRANSID** command in [IBM Knowledge Center](#).

2.3.5 Passing data with CICS channel and containers

In most cases, a child task runs to execute business logic or call a service on behalf of a parent task and data needs to be passed to the child task and returned to the parent task. CICS has an established method for data passing using CICS channel and containers. The asynchronous API commands use channels, while maintaining flexibility, with optional CHANNEL parameters on the **RUN TRANSID** and **FETCH** commands.

Recap of CICS channel and containers: Structured data (akin to COMMAREA data without the 32 K limit) resides in named *containers*. The container names (and data formats) are known to both the sender and receiver of containers. One or many containers are grouped together in to *channels*. The name of the channel might be important to the sender and receiver. A program that is begun with a channel will also reply with that channel.

Passing a channel from a parent to a child task

A parent can name a channel to pass to a child task by specifying the CHANNEL parameter on the **RUN TRANSID** command. To maintain data integrity, the containers are copied and made available to the child task in a channel of the same name, which will be the current channel for the child task.

Note on data integrity: When data is being passed from parent to child, it is important that no two tasks can operate on the data at the same time. In synchronous LINK situations, the caller is suspended until the task being called completes, so there is no risk of two parties trying to modify the data simultaneously. In asynchronous environments, it is important to pass data by copy rather than by reference to ensure data integrity.

In practice, this means that by copying the channel from the parent to the child, there is no possibility of the parent updating the input whilst in the possession of the child. Also, a parent can pass the “same channel” to multiple child tasks, and each will have their own instance of the channel. Child tasks cannot affect other child tasks channels.

Figure 2-2 shows a parent task issuing two **EXEC CICS RUN TRANSID** commands with the **CHANNEL** parameter, to run two child tasks. The parent task has a channel called *A*. On each of the **RUN TRANSID** commands, channel *A* is copied into new instances of channels (also named *A*) for the current channel of each child task.

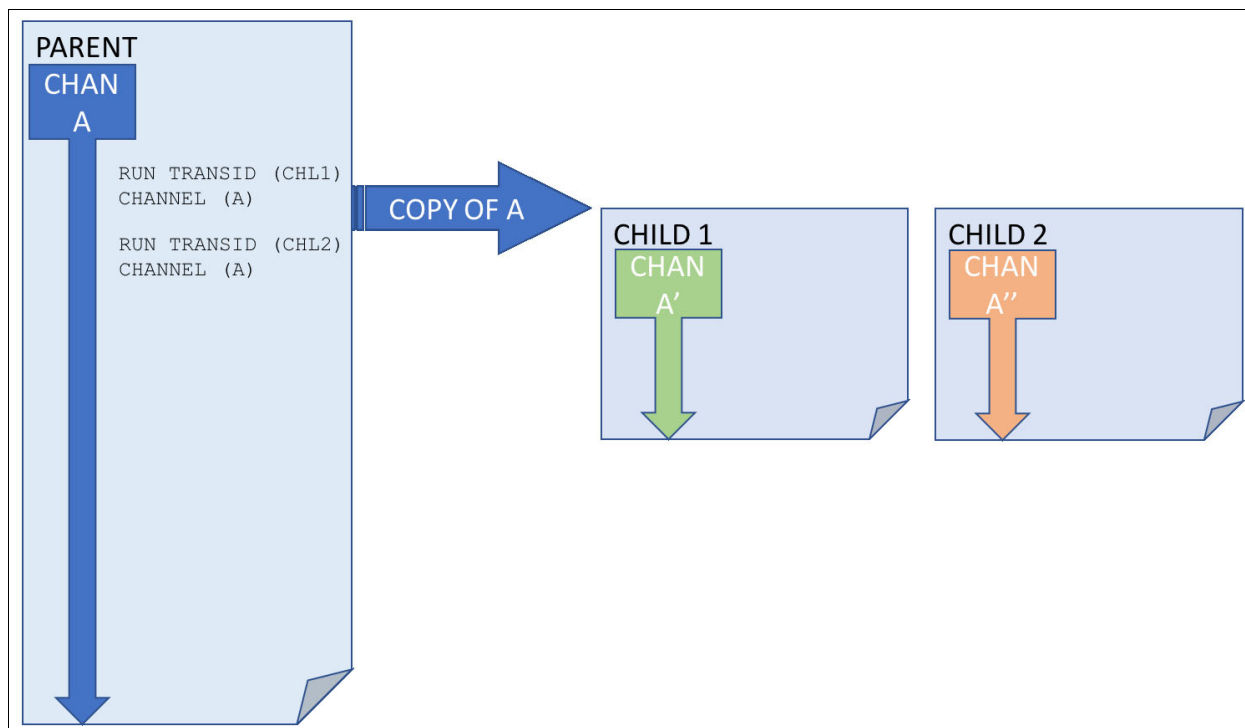


Figure 2-2 A parent task running two child tasks with a channel

Working with containers in a child task

No special processing is required from the child task to work with the containers in a channel that is run using the asynchronous API command.

A child task uses standard CICS API commands to work with containers on the channel, just as a regular CICS program would. The child task's current channel is named and contains the same containers as the channel specified on the parent's **RUN TRANSID** command.

Returning data in containers from a child to parent

A child task's current channel contents becomes available to the parent after the child task terminates. Thus, a child updates the contents of the channel with data it intends to make available back to the parent.

Due to the asynchronous nature of parent and child tasks, it is likely that a child will complete ahead of the parent requesting the resulting channel from the child. CICS manages the channel from completed child tasks until the parent is ready to consume the results. CICS cleans up (discards) the child channel and associated container data, in situations where the parent task has already terminated, when the child task terminates.

Parent fetching results from a completed child

When a parent task is ready to consume the results from a child task, the parent issues a **FETCH CHILD** or **FETCH ANY** command, specifying the optional **CHANNEL** parameter. The parent specifies a variable or data area in the **CHANNEL** parameter, because on a successful **FETCH** command, the API returns a new name for the returned channel. It will not be the same name as initially specified on the **RUN TRANSID** command.

The reason for a channel being returned with a different name is again for data integrity. We do not want data in an existing channel being overwritten by fetching a child's channel with the same name. Also, if two child tasks were run with the same initial channel name, we would not want them to overwrite each other when being fetched back. By providing new unique names, both child task's channels can be independently referenced.

Figure 2-3 shows a parent task issuing two **EXEC CICS FETCH CHILD** commands with the **CHANNEL** parameter, to retrieve the results from two child tasks. On each of the **FETCH CHILD** commands, the parent specifies a variable to contain the unique name of the channel from the child task. After the two **FETCH CHILD** commands, three channels are bound to the parent task:

- ▶ The original channel *A*
- ▶ Channel from *CHIL1*
- ▶ Channel from *CHIL2*

Each channel has a unique name and can be specified on container API commands, such as **EXEC CICS GET CONTAINER(x) CHANNEL(ch12Chan)**.

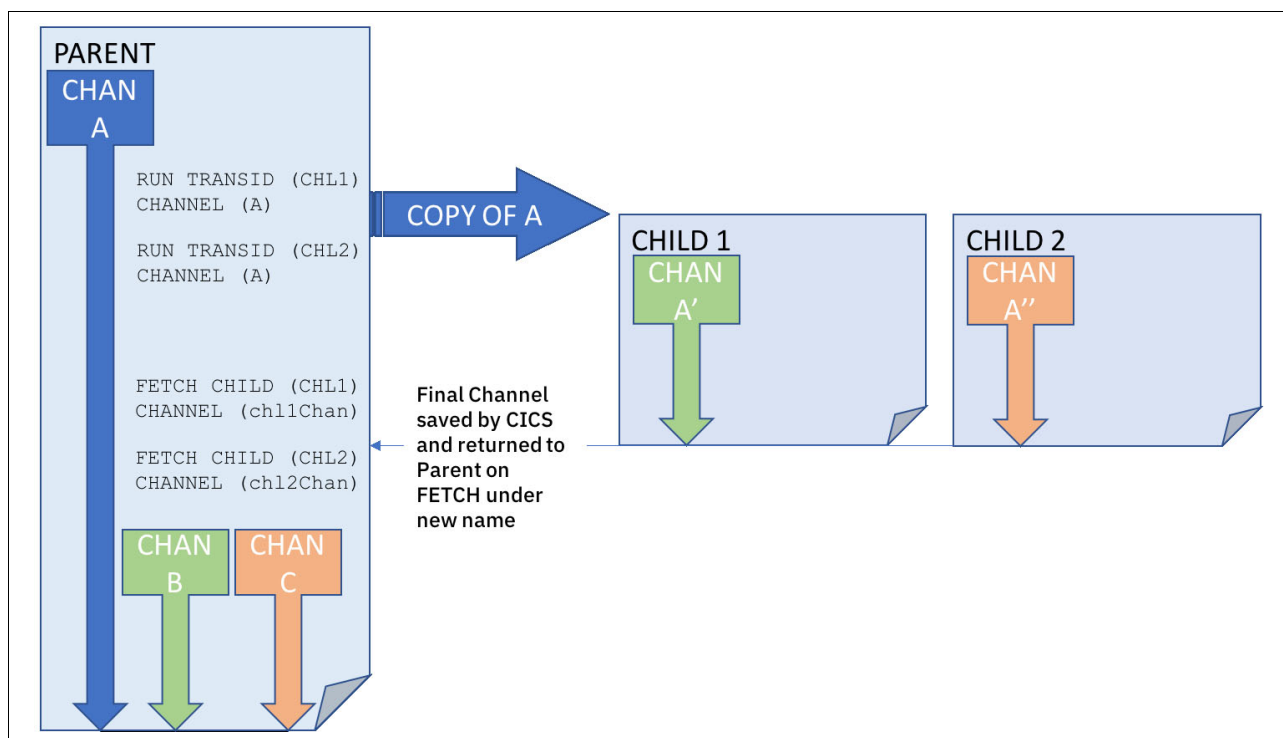


Figure 2-3 A parent task fetching two child tasks with a channel

It is perfectly valid for a parent to not fetch a completed child's channel data. CICS maintains the channel, in case the parent fetches the results in future. The channel is automatically discarded if the parent completes without fetching the channel or if the parent issues a **FREE CHILD** command.

If a parent has fetched a child task's channel, the fetched channel is bound to that parent program's link level. After it is fetched, if the parent issues a **FREE CHILD** command, the channel will not be deleted, because it is bound to the parent. The parent can discard the channel by issuing the **DELETE CHANNEL** command. Alternatively, the fetched channel is discarded by the task termination processing by CICS automatically when the parent task completes.

Tip: If you expect a child task to return data to a parent via a channel, always specify the **CHANNEL** parameter on the **RUN TRANSID** command, even if you have no data to pass to the child. This process ensures that the child task is created with a channel input/output setup.

You can simply specify any channel name (a channel name that does not exist) on the **RUN TRANSID** command, and the API creates the channel and makes it available to the child.

DFHTRANSACTION channel

Channels are normally limited to the scope of a link level. One exception is DFHTRANSACTION, which is still visible at different link levels within a transaction's scope. You can specify the DFHTRANSACTION channel on the **RUN TRANSID** command, which copies the contents to the child task. However, the **FETCH** command receives a regular link-level scoped channel on return.

COMMAREA communication

A great feature of the asynchronous API is to run existing business logic as an asynchronous child task. CICS users often have a lot of investment in applications that use COMMAREAs for data passing. Although the asynchronous API does not support COMMAREAs on the API calls, it is possible to create a wrapper program that accepts a channel and converses with exiting assets via a COMMAREA. This topic is explored further in Chapter 7, "Tips and tricks" on page 95.

2.3.6 CICS Asynchronous Services domain

CICS TS V5.4 introduced a new domain called the Asynchronous Services (AS) domain. In the CICS domain architecture, AS domain is responsible for the management of asynchronous patterns created by using the asynchronous API. Other domains also play a part in the overall management, such as for channels and transactions.

AS domain manages the relationship between parent and child tasks. The domain also handles the generation and deletion of control blocks when tasks are initiated and terminated.

There are rules for when state is deleted, so asynchronous patterns might seem as though they maintain storage for a longer amount of time than you expect. Typically, this storage is maintaining channel data and relationship control blocks for longer than the lifetime of child tasks, in case a parent requests the results later.

Typically, the philosophy of "the last one out closes the lights" holds true with state clean-up, which is often the parent task's termination. However, in scenarios with a long-running parent you might see an accumulation of control blocks and channels. In these scenarios, use the **FREE CHILD** command to reduce resource usage. It is not commonly required for scenarios in which the parent task completes in a timely manner.

Another area of management in CICS and AS domain is the protection against running too many child tasks. If the system reaches MAXTASK conditions, the AS domain suspends the initiation of new child tasks. When the system begins to recover, the queued child tasks begin to be dispatched. This action is intended as a system fail safe and not as a standard control mechanism.

The AS domain is also mentioned in later chapters with debugging, management, and tracking tasks.

2.3.7 Timeouts

By default, a parent task issuing a **FETCH CHILD** or **FETCH ANY** command suspends until the named child, or at least one child, completes. This process is sensible behavior if a response from the child is required for the parent to continue execution and if the child is reliable.

If you are calling unreliable services, or perhaps attempting to keep within an SLA, and do not want to suspend indefinitely, use the **TIMEOUT** parameter on the **FETCH CHILD** or **FETCH ANY** commands. If the timeout duration is met before the child completes, the command returns a non-zero response code (a NOTFINISHED RESP code and a 53 RESP2 code), and the parent task resumes execution.

Note: The **TIMEOUT** parameter on the **FETCH** command indicates the length of time the parent is willing to wait for a response. It does not affect the execution of the child task. Following a timed-out response on a **FETCH** command, the child continues to execute (for example, the timeout will not cancel nor abend a child task).

It is possible, for example, a parent to reissue the **FETCH** command. Example 2-5 illustrates pseudocode of how you can retry to **FETCH** the results of an unpredictable service.

Example 2-5 Pseudocode illustrating how to retry to FETCH results of an unpredictable service

```
//An attempt is made to fetch the results of a child in a timely manner
EXEC CICS FETCH CHILD(mychild) CHANNEL(mychan) TIMEOUT(2000)
RESP(resp) RESP2(resp2);

//If timed out then send a courtesy message and do something else
If (resp=NOTFINISHED and resp2=53) then do
  DISPLAY "In Progress, please wait...";

  ... Do some other work ...

//Finally retrieve child's results
EXEC CICS FETCH CHILD(mychild) CHANNEL(mychan)
  RESP(resp) RESP2(resp2);
```

It is possible to specify zero (0) on the **TIMEOUT** parameter. This special value indicates that a timeout is not being specified. It is analogous to not specifying the parameter at all and results in the default behavior that the parent suspends until the child task completes.

If you want to issue the **FETCH** command without blocking the parent (even if the child has not completed), specify the **NOSUSPEND** option.

The **TIMEOUT** parameter on the **FETCH CHILD** and **FETCH ANY** commands is complimentary to any other timeout settings you might have. If the workload is subject to other timeouts and settings, such as **DTIMEOUT**, they continue to behave as expected.

2.4 Considerations for using the CICS asynchronous API

This section highlights some general considerations in using the CICS asynchronous API. It offers guidance about suggested approaches on using the CICS asynchronous API to provide a solid foundation for your CICS applications. However, it is only guidance. Your environments and applications might vary and might benefit from different approaches.

2.4.1 Child GETs and parents UPDATE

Child tasks should perform **GETs** of data and leave the **UPDATE** actions to the parent task.

As previously mentioned, parent and child tasks are separate units of work (UOW). If, for example, a parent abends, it can back-out its own work but will be unconnected from the execution of the child tasks. The child tasks might be waiting to be dispatched, actively executing, or even completed. If the child tasks are limited to **GET** actions, they can be safely discarded without compromising the state of the data.

As full transaction environments, it is possible for child tasks to perform **UPDATE** actions on data. However, you might be required to code compensation logic to revert changes, if the need arises.

2.4.2 Allow the same parent program to run and fetch child tasks

It is possible for a parent program to use the **RUN TRANSID** command and for another program in the same transactional scope to fetch the child tasks result. Indeed, a strength of CICS is its multi-language support. However, just because you can start child tasks in a COBOL program and then fetch them in a Java program does not mean that you should!

It is advised for manageability that the parent program that begins a child task remain the program that fetches the results from the child task. This process is particularly useful if you are using the **FETCH ANY** command, because it can become difficult to match the **RUN TRANSID** to **FETCH ANY** commands if they reside in different programs. Also note that fetched channels are scoped to a link level. So passing results can be problematic if it is not the parent that fetches the child's results.

2.4.3 Long-running parents should use the FREE CHILD command

In a typical short-lived application, the channel and control blocks are deleted by CICS as the tasks terminate. In a long-running parent application, for example a daemon process that exists to begin many child tasks, there can be a build-up to control blocks as child tasks complete. In addition, there can be a build-up of sizeable channel data. Without task termination, the CICS system cannot safely discard this data.

It is advisable that a long-running parent task deletes any fetched channels and issues **FREE CHILD** commands against child tasks that are no longer required.

2.4.4 Keep track of fetched channels

Data is passed between parent and child tasks using CICS channel and containers. To return data from a child, the child task updates containers on its current channel. During the child tasks termination, the channel is managed or owned by CICS (AS domain). The parent task can retrieve the child's channel by specifying the CHANNEL parameter on the **FETCH CHILD** or **FETCH ANY** commands.

Specifying the CHANNEL parameter on a **FETCH** command is more active than a simple inquiry. The channel will bind to the issuing parent. Thus, the ownership changes from the AS domain to the parent task. Any subsequent **FETCH** command for the same child's channel fails because it is now in the control of the parent task, not the AS domain.

Save returned channel names to unique variables in your application logic for future reference. This process is particularly important when using looping algorithms or writing generic functions, such as issuing the **EXEC CICS FETCH ANY CHANNEL** command.

An alternative mechanism, if searching through many completed child tasks, is to issue **EXEC CICS FETCH ANY** commands (without the CHANNEL parameter). Then, when you identify which child you intend to process, issue the **EXEC CICS FETCH CHILD CHANNEL** command, to bind only channels to the parent when needed.

2.4.5 Review MAXTASK and set transaction classes

Application developers have had the ability to begin additional transactions with the use of the **START** API commands. However, having to architect the framework for asynchronous processing patterns means that this capability has not always been used. Thus, the simplicity of the asynchronous API might generate more transactions in a CICS system.

It is advised that the settings for the MAXTASK parameter be reviewed.

There are many ways to limit the amount of work entering a CICS region. At the crudest level, you can limit an applications interface to requesters who are known to you and only call the services “*n*” times a day. Other mechanisms are software and hardware appliances that control the workflow into a CICS region.

Whatever your mechanism, it is advised to have controls in place to limit the amount of work that a region accepts that cause tasks to be run asynchronously.

One such mechanism is to put asynchronous application parents into a transaction class by using the TRANCLASS parameter, thereby controlling the number of parent tasks (and thus the maximum number of children they can run).

Note: Do not specify parent transactions and child transactions in the same transaction class. An unfortunate scenario is that the entire class is full of parent tasks trying to run a child task each but not being able to, thus deadlocking the transaction class.

For further details, see 9.4, “Policing parent tasks with CICS policy” on page 155.

2.4.6 Parameterizing timeouts

In many cases, the results of a child are needed for the parent to continue their business logic. In other situations, and when the responsiveness of the application is important, it is possible for a parent to specify a TIMEOUT parameter on the **FETCH** command.

If you require a timeout, or if there is a possibility that you might need to in future, code the TIMEOUT parameter on the **FETCH** command. The value of the timeout should be parameterized, such as being read from a file or database table, earlier in the application.

By inserting the TIMEOUT parameter and parameterizing the timeout value, you prevent the need for a future code update or recompilation. If it transpires that a timeout is not required, the special value of zero can be specified to indicate a timeout should not be enforced.

See Chapter 6, “Creating a Java-based controller in a mixed-language environment” on page 73 for a worked example of adding a parameterized timeout to a **FETCH CHILD** command.



Extending applications while minimizing the impact to response time

This chapter walks through one of the most common scenarios for using the CICS asynchronous API by extending an application while minimizing the impact to its response time.

Applications evolve over time. As they evolve, you typically need to add functionality as users demand more intelligent behaviors. This need means that applications must now request data from more sources and use this data to perform functions in a more informed and “intelligent” way. In the case of CICS applications, this process usually involves adding new requests for additional data and services.

When increasing the number of requests in an application, the immediate concern is one of response time. By adding more work to the application or by making the application wait on external requests, its execution time is increased, which causes the following issues:

- ▶ The increased execution time is undesirable to demanding users who want more intelligent applications but also want these applications to respond faster.
- ▶ With each invocation of the application now taking longer, there is a much higher demand on system resources as the number of concurrent requests in progress in the system grows. This issue is exacerbated further as use of the application increases due to its now enhanced functionality.

The scenario in this chapter demonstrates how you can use the CICS asynchronous API to minimize the impact on response time when adding new functionality to applications.

3.1 Overview of the scenario

Important process and content information: This chapter includes a series of actions to successfully complete the described scenario. Be aware that the steps that you need to complete for this scenario are included in numbered paragraphs. Although the numbered steps might occur in different sections throughout the chapter, you still need to complete the steps in the order in which they occur.

This scenario, and those over the next few chapters, use a sample application to illustrate the necessary concepts. You can find the source code for the complete application, along with the setup instructions, in the [GitHub](#) repository.

Tip: The repository contains the final version of the code after all of the scenarios in this book are applied. If you want to follow along with this chapter, start with an earlier version of the code at this commit: [Chapter 3 Start Tag](#).

3.1.1 Description of the sample application

The application is written in Common Business Oriented Language (COBOL) and provides summary data for a fictional web (mobile) banking home page (the WEBHOME program). The main logic is shown in Example 3-1.

Example 3-1 The WEBHOME program

```
* First step is to retrieve the account number
  PERFORM GET-INPUT-ACCOUNT-NUMBER

* ----
* Create the input container for children to access
* ----
      EXEC CICS PUT CONTAINER ( INPUT-CONTAINER )
                        FROM   ( ACCOUNT-NUMBER-IN )
                        CHANNEL ( MYCHANNEL )
                        RESP    ( COMMAND-RESP )
                        RESP2   ( COMMAND-RESP2 )

      END-EXEC

      PERFORM CHECK-COMMAND

* ----
* Get the customers name
* ----
      EXEC CICS LINK PROGRAM ( GET-NAME )
                        CHANNEL ( MYCHANNEL )
                        RESP    ( COMMAND-RESP )
                        RESP2   ( COMMAND-RESP2 )

      END-EXEC

      PERFORM CHECK-COMMAND

      EXEC CICS GET CONTAINER ( GETNAME-CONTAINER )
                        CHANNEL ( MYCHANNEL )
```

```

                                INTO    ( CUSTOMER-NAME )
                                RESP    ( COMMAND-RESP )
                                RESP2   ( COMMAND-RESP2 )
END-EXEC

PERFORM CHECK-COMMAND

INITIALIZE STATUS-MSG
STRING 'Welcome '
      DELIMITED BY SIZE
      CUSTOMER-NAME
      DELIMITED BY SPACE
      INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

* ----
* Get the customers current account details
* ----
EXEC CICS LINK PROGRAM ( ACCTCURR )
              CHANNEL ( MYCHANNEL )
              RESP    ( COMMAND-RESP )
              RESP2   ( COMMAND-RESP2 )
END-EXEC

PERFORM CHECK-COMMAND

EXEC CICS GET CONTAINER ( ACCTCURR-CONTAINER )
              CHANNEL ( MYCHANNEL )
              INTO    ( CURRENT-ACCOUNTS )
              RESP    ( COMMAND-RESP )
              RESP2   ( COMMAND-RESP2 )
END-EXEC

PERFORM CHECK-COMMAND

PERFORM PRINT-CURRENT-ACCOUNTS-DETAILS

* Send a message to the screen to
* notify terminal user of completion
  MOVE 'COMPLETE' TO CURRENT-STATUS
  PERFORM PRINT-TEXT-TO-SCREEN

* Display a conclusion message that also includes a timestamp
  INITIALIZE STATUS-MSG
  MOVE 'Ended Web banking log-on data retrieval' TO MSG-TEXT
  PERFORM PRINT-STATUS-MESSAGE

* Return at end of program
  EXEC CICS RETURN
  END-EXEC
.
```

The main program, `WEBHOME.cb1`, takes a customer number as input and returns data about that customer's accounts along with the customer's name. The requests for the customer name and account details are performed synchronously by linking to the following programs:

- ▶ `GETNAME`
- ▶ `ACCTCURR`

To keep the sample simple, the data returned is hard coded into each program. However, in the real world, this data would have been requested from a database or remote service. To simulate this type of processing, each program delays for a few seconds before returning. This delay represents the time that it would take for the data to be collected or calculated in a real-world environment. The delays are exaggerated from real-world timings, which likely would be subsecond.

The specific timing of the delays becomes more important in later chapters. However, for the scenario in this chapter, they provide the application with a perceivable overall execution time with which to illustrate asynchronous techniques.

Figure 3-1 shows how the `GETNAME` and `ACCTCURR` programs contribute to the overall execution time of the `WEBHOME` program.

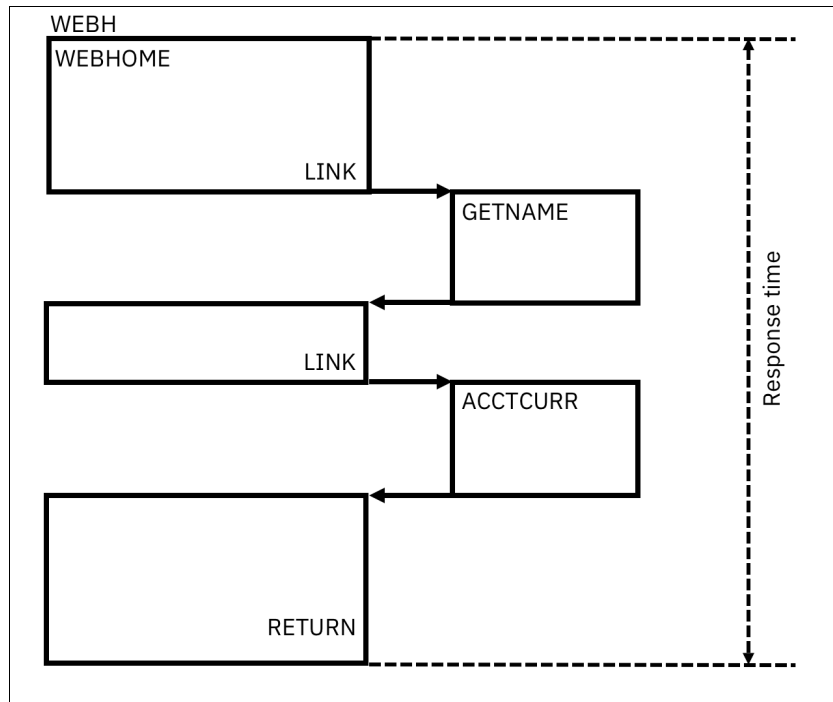


Figure 3-1 Application execution time

You can try the `WEBHOME` program by running the `WEBH` transaction from the terminal and by passing the customer number `0001`.

1. Issue the following transaction and customer number in the CICS terminal window:
WEBH 0001

The WEBHOME program executes and displays status messages in the CICS language environment messages log, such as CEEMSG, as shown in Example 3-2. Here you can see the name and account details for the given customer number.

Example 3-2 WEBH transaction results

```

TC56WEBH 20171011152321 15:23.21 Started Web banking log-on data retrieval
TC56WEBH 20171011152325 15:23.25 Welcome Pradeep Gohil
TC56WEBH 20171011152328 15:23.28 Acc: 20140720 Bal: £0.01      Overdraft: £0.00
TC56WEBH 20171011152328 15:23.28 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
TC56WEBH 20171011152328 15:23.28 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
TC56WEBH 20171011152328 15:23.28 Ended Web banking log-on data retrieval

```

3.1.2 Objective of the scenario

Let's imagine that this application needs to be updated to also provide account data for accounts that this customer has with a partner bank. To achieve this a new program, ACCTPTNR must be called to retrieve the data. Similar to the GETNAME and ACCTCURR programs, the ACCTPTNR program returns hard-coded data but delays for a few seconds to simulate a real request to a partner bank's system.

If you were to link to this new program synchronously, as with the other two programs, you would inevitably end up increasing the overall response time of the application by adding the execution time for the ACCTPTNR program to the end of the application. The resulting slower response time is undesirable to users and might even take the application outside of the maximum allowed response time for its Service Level Agreement (SLA), as shown in Figure 3-2.

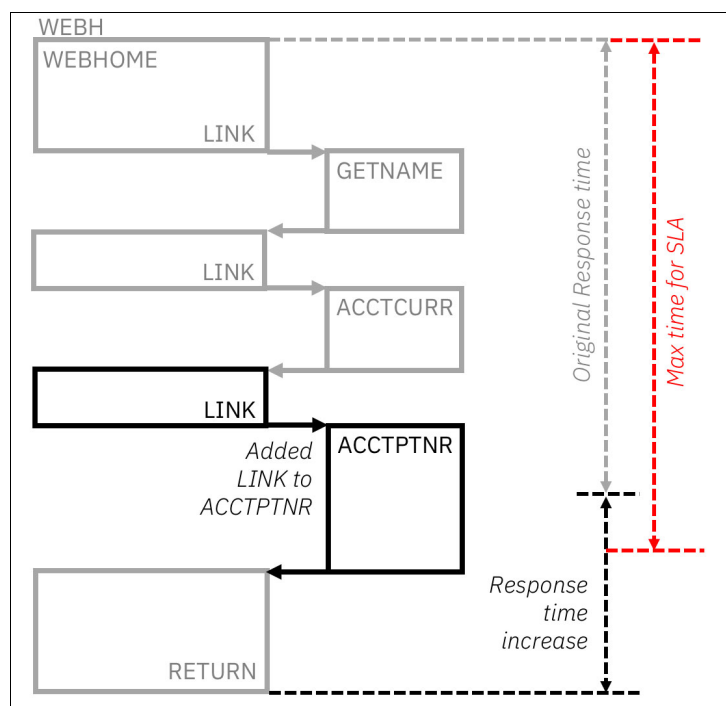


Figure 3-2 Adding the ACCTPTNR program execution time to the WEBHOME program synchronously

The objective is to add a request to retrieve the account details from the partner bank without impacting the response time of the overall application.

3.2 Add a new request using the CICS asynchronous API

The CICS asynchronous API provides a way for CICS programs to run CICS transactions and to fetch the results at a later time. You can use this API to update the WEBHOME program with a call to ACCTPTNR without impacting its overall response time.

To add a new request:

- ▶ Define a transaction, PTNR, to run the ACCTPTNR program.
- ▶ Add logic to print the partner account details.
- ▶ Add a call to the **RUN TRANSID** command at the start of the WEBHOME program to run the PTNR transaction.
- ▶ Add a call to the **FETCH CHILD** command at the end of the WEBHOME program to collect the results from PTNR.

Tip: If you are following this scenario using the sample on GitHub, you can find the [changes from this chapter](#) online on GitHub also.

By starting the PTNR transaction at the beginning of the WEBHOME program and fetching its results at the end (as shown in Figure 3-3), the time that the ACCTPTNR program takes to run is effectively negated, because it happens simultaneously to the work that the WEBHOME program was already doing.

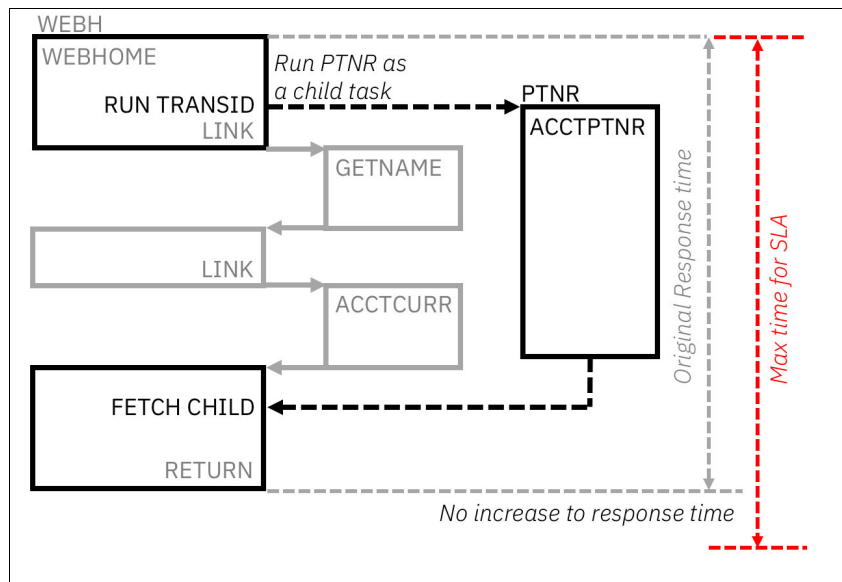


Figure 3-3 Calling the ACCTPTNR program asynchronously

3.2.1 Defining the PTNR transaction to run ACCTPTNR

Because the CICS asynchronous API operates on *transactions* not programs, you must first define a transaction that you will use to run the ACCTPTNR program. This transaction can have any name, but for the purposes of this scenario, we use PTNR as the transaction name.

2. Ensure that a TRANSACTION definition is defined in CICS with the attributes shown in Example 3-3.

Example 3-3 PTNR TRANSACTION definition attributes

```
TRANSACTION(PTNR)
GROUP(ASYNCAPI)
PROGRAM(ACCTPTNR)
DESCRIPTION(Transaction to request partner bank account details as part of the
CICS asynchronous API Redbook sample)
```

3.2.2 Adding logic to print the partner account details

To print the results from the ACCTPTNR program, add the PRINT-PARTNER-ACCOUNTS-DETAILS paragraph to process the results.

3. Add the code in **bold** font shown in Example 3-4 to the PRINT-STATUS-MESSAGE paragraph towards the end of WEBHOME.cbl.

Example 3-4 PRINT-PARTNER-ACCOUNTS-DETAILS paragraph

```
* Print partner account details
PRINT-PARTNER-ACCOUNTS-DETAILS.
  IF NUMBER-OF-ACCOUNTS OF PARTNER-ACCOUNTS > 0 THEN
    MOVE 1 TO COUNTER
    PERFORM UNTIL COUNTER >
      NUMBER-OF-ACCOUNTS OF PARTNER-ACCOUNTS
      INITIALIZE STATUS-MSG
      STRING 'Acc: '
        DELIMITED BY SIZE
        ACCT-NUMBER OF PARTNER-ACCOUNTS (COUNTER)
        DELIMITED BY SPACE
        ' Bal: $'
        DELIMITED BY SIZE
        BALANCE OF PARTNER-ACCOUNTS (COUNTER)
        DELIMITED BY SIZE
        ' Overdraft: $'
        DELIMITED BY SIZE
        OVERDRAFT OF PARTNER-ACCOUNTS (COUNTER)
        DELIMITED BY SIZE
      INTO MSG-TEXT
      PERFORM PRINT-STATUS-MESSAGE
      ADD 1 TO COUNTER
    END-PERFORM
  END-IF
  .

* Print status message
PRINT-STATUS-MESSAGE.
```

Later examples in 3.2.4, “Adding the FETCH CHILD command to the WEBHOME.cbl program” on page 35 use PRINT-PARTNER-ACCOUNTS-DETAILS to process the returned data from the ACCTPTNR program.

3.2.3 Adding the RUN TRANSID command to WEBHOME.cbl

To call the newly defined PTNR transaction from the WEBHOME program, use the **EXEC CICS RUN TRANSID** command. This command runs the PTNR transaction as a child task of the WEBH

transaction and allows the WEBHOME program to continue processing instead of waiting for the ACCTPTNR transaction to complete. (The next section looks at fetching the results from this child task.)

You must first add data field definitions for ACCTPNTR-TRAN and ACCTPTNR-TKN to WEBHOME.cbl.

4. Add the code in **bold** font in Example 3-5 to the data definitions of WEBHOME.cbl.

Example 3-5 Data definitions for ACCTPTNR-TRAN and ACCTPTNR-TKN

```

1 PROGRAM-NAMES.
    2 GET-NAME                PIC X(8) VALUE 'GETNAME '.
    2 ACCTCURR                PIC X(8) VALUE 'ACCTCURR'.
    2 ACCTPTNR                PIC X(8) VALUE 'ACCTPTNR'.
    2 GETLOAN                 PIC X(8) VALUE 'GETLOAN '.

    1 TRANSIDS.
    2 ACCTPTNR-TRAN           PIC X(4) VALUE 'PTNR'.

    1 CHILD-TOKENS.
    2 ACCTPTNR-TKN           PIC X(16).

    1 CHILD-RETURN-STATUS     PIC S9(8) USAGE BINARY.
    1 CHILD-RETURN-ABCODE     PIC X(4).

```

ACCTPTNR-TRAN is set to a value of 'PNTR' and is used as the transaction ID.

ACCTPTNR-TKN is used to store the child token that we will use later to fetch the results. See 3.2.4, “Adding the FETCH CHILD command to the WEBHOME.cbl program” on page 35.

5. Next, add the code in **bold** in Example 3-6 to WEBHOME.cbl before the **LINK** to GETNAME.

Example 3-6 Running PTNR asynchronously

```

* -----
* Asynchronously run PNTR to get account details
* from the partner bank
* -----
    EXEC CICS RUN TRANSID ( ACCTPTNR-TRAN )
                      CHANNEL ( MYCHANNEL )
                      CHILD  ( ACCTPTNR-TKN )
                      RESP   ( COMMAND-RESP )
                      RESP2  ( COMMAND-RESP2 )

    END-EXEC

    PERFORM CHECK-COMMAND

* ----
* Get the customers name
* ----
    EXEC CICS LINK PROGRAM ( GET-NAME )

```

Note: For convenience, the CHECK-COMMAND paragraph is already provided to check the response codes of any EXEC CICS commands (using the COMMAND-RESP and COMMAND-RESP2 fields).

There are a few important concepts to understand at this stage:

- ▶ Placing the call for the **RUN TRANSID** command
- ▶ Passing data to the child task

Placing the call for the **RUN TRANSID** command

First, the position of the **RUN TRANSID** command the WEBHOME program in the WEBHOME program logic is most important. Because the child task is run asynchronously, the **RUN TRANSID** command returns immediately. This process allows the parent program, WEBHOME, to continue its remaining processing without having to wait for the child task to complete. By calling the **RUN TRANSID** command as early as possible, *before* the parent program performs any of its existing logic, you maximize the benefit of running the PTNR transaction asynchronously. This method effectively ensures that the most amount of concurrent processing occurs, resulting in the biggest response time savings, as illustrated in Figure 3-4.

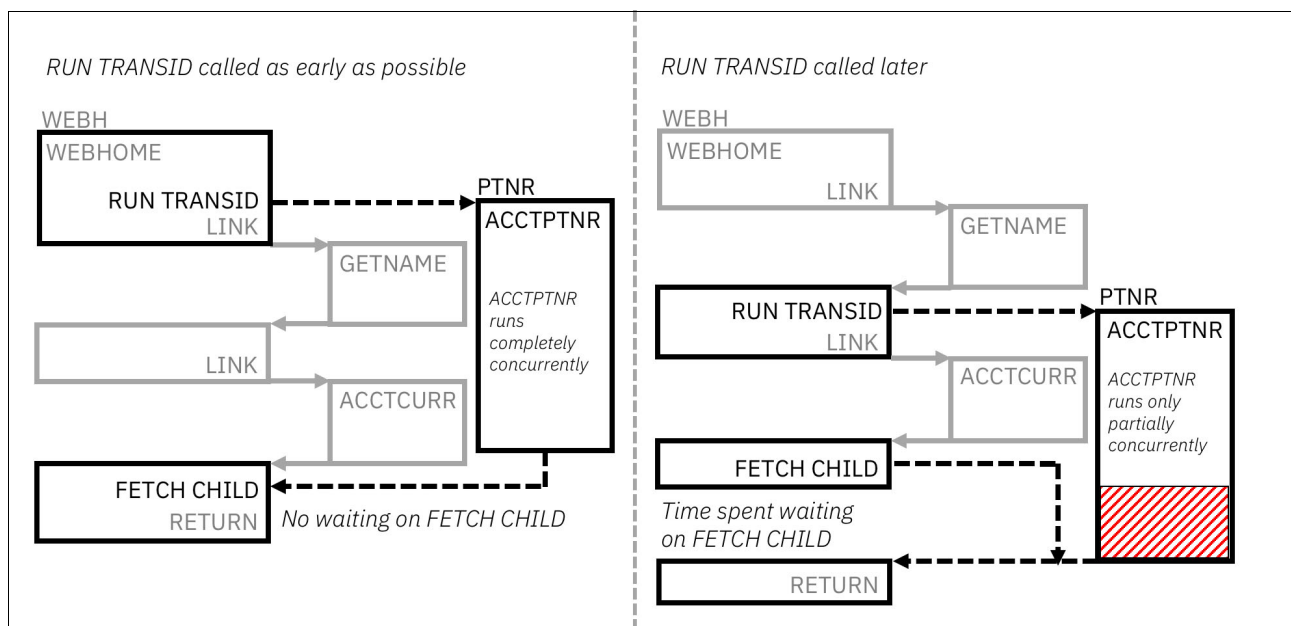


Figure 3-4 Comparing response time savings for early versus late asynchronous calls

Passing data to the child task

The second point to understand is how data is passed to the PTNR transaction. The **RUN TRANSID** call uses the optional CHANNEL parameter to pass data to the child task. The ACCTPTNR program uses the same interface as the GETNAME and ACCTCURR programs. Thus, you can reuse the channel, MYCHANNEL, and container, INPUTCONTAINER, that are already set up to pass the customer number. There are no concurrency issues to worry about because the **RUN TRANSID** command always takes a copy of the channel and its containers to give to the child task. The copy is made when the **RUN TRANSID** command is issued.

3.2.4 Adding the **FETCH CHILD** command to the WEBHOME.cbl program

Now that the WEBHOME program runs the PTNR transaction asynchronously as a child task, the final step is to add code to fetch and render the results. You get the results from the PTNR transaction using the **FETCH CHILD** command by passing the ACCTPTNR-TKN child token that was returned from the **RUN TRANSID** call. Use the PRINT-PARTNER-ACCOUNTS-DETAILS paragraph created in 3.2.2, “Adding logic to print the partner account details” on page 33 to print the partner account details.

You must first add data field definitions for ACCTPTNR-CHAN and ACCTPTNR-CONTAINER to the WEBHOME.cb1 program.

6. Add the code shown in **bold** font in Example 3-7 to the data definitions of the WEBHOME.cb1 program.

Example 3-7 Data definitions for ACCTPTNR-CONTAINER and ACCTPTNR-CHAN

```

1 CONTAINER-NAMES.
  2 INPUT-CONTAINER      PIC X(16) VALUE 'INPUTCONTAINER  '.
  2 GETNAME-CONTAINER    PIC X(16) VALUE 'GETNAMECONTAINER'.
  2 ACCTCURR-CONTAINER    PIC X(16) VALUE 'ACCTCURRCONT  '.
  2 ACCTPTNR-CONTAINER    PIC X(16) VALUE 'ACCTPTNRCONT  '.

...

1 CHILD-TOKENS.
  2 ACCTPTNR-TKN          PIC X(16).

1 RETURN-CHANNELS.
  2 ACCTPTNR-CHAN          PIC X(16).

```

ACCTPTNR-CHAN is defined as a PIC X(16) field and is used to store the name of the reply channel of the child task. The name is generated by CICS and is unique within the scope of the current link level.

ACCTPTNR-CONTAINER is set to 'ACCTPTNRCONT' and is the name of the return container for the ACCTPTNR program.

7. Next, add the code shown in **bold** font in Example 3-8 to the WEBHOME.cb1 program before it notifies the terminal user of completion.

Example 3-8 Fetching the results from the PTNR child task

```

* -----
* Get the customers current account details from the
* partner bank
* -----

      EXEC CICS FETCH CHILD      ( ACCTPTNR-TKN )
                        CHANNEL    ( ACCTPTNR-CHAN )
                        COMPSTATUS ( CHILD-RETURN-STATUS )
                        ABCODE     ( CHILD-RETURN-ABCODE )
                        RESP       ( COMMAND-RESP )
                        RESP2      ( COMMAND-RESP2 )

      END-EXEC

      PERFORM CHECK-COMMAND
      PERFORM CHECK-CHILD

      EXEC CICS GET CONTAINER ( ACCTPTNR-CONTAINER )
                        CHANNEL ( ACCTPTNR-CHAN )
                        INTO    ( PARTNER-ACCOUNTS )
                        RESP     ( COMMAND-RESP )
                        RESP2    ( COMMAND-RESP2 )

      END-EXEC

      PERFORM CHECK-COMMAND

```

PERFORM PRINT-PARTNER-ACCOUNTS-DETAILS

- * Send a message to the screen to
 - * notify terminal user of completion
- ```
MOVE 'COMPLETE' TO CURRENT-STATUS
PERFORM PRINT-TEXT-TO-SCREEN
```
- 

**Note:** For convenience, the CHECK-CHILD paragraph is provided to check the completion status and to abend code for any fetched child tasks by using the CHILD-RETURN-STATUS and CHILD-RETURN-ABCODE fields. These fields are provided as follows:

- ▶ CHILD-RETURN-STATUS is defined as PIC S9(8) USAGE BINARY. It is used to store a CVDA value indicating the completion status of the child task.
- ▶ CHILD-RETURN-ABCODE is defined as PIC X(4). It is used to store the abend code of the child task if it terminated abnormally.

There are a few important concepts to understand at this stage:

- ▶ Placement of the call for the **FETCH CHILD** command
- ▶ Use of the child token
- ▶ Checking completion status of the child task
- ▶ Retrieving data from the child task

### Placing the call for the **FETCH CHILD** command

First, the position of the **FETCH CHILD** command in the WEBHOME program logic is most important. Because the child task is run asynchronously and might still be running, the **FETCH CHILD** command waits and returns after the child task is complete. By calling the **FETCH CHILD** command as late as possible, *after* the WEBHOME program has completed its existing logic, you minimize the idle time spent waiting for the PTNR transaction to complete. This method effectively ensures that the most amount concurrent processing occurs, resulting in the biggest response time savings, as illustrated in Figure 3-5.

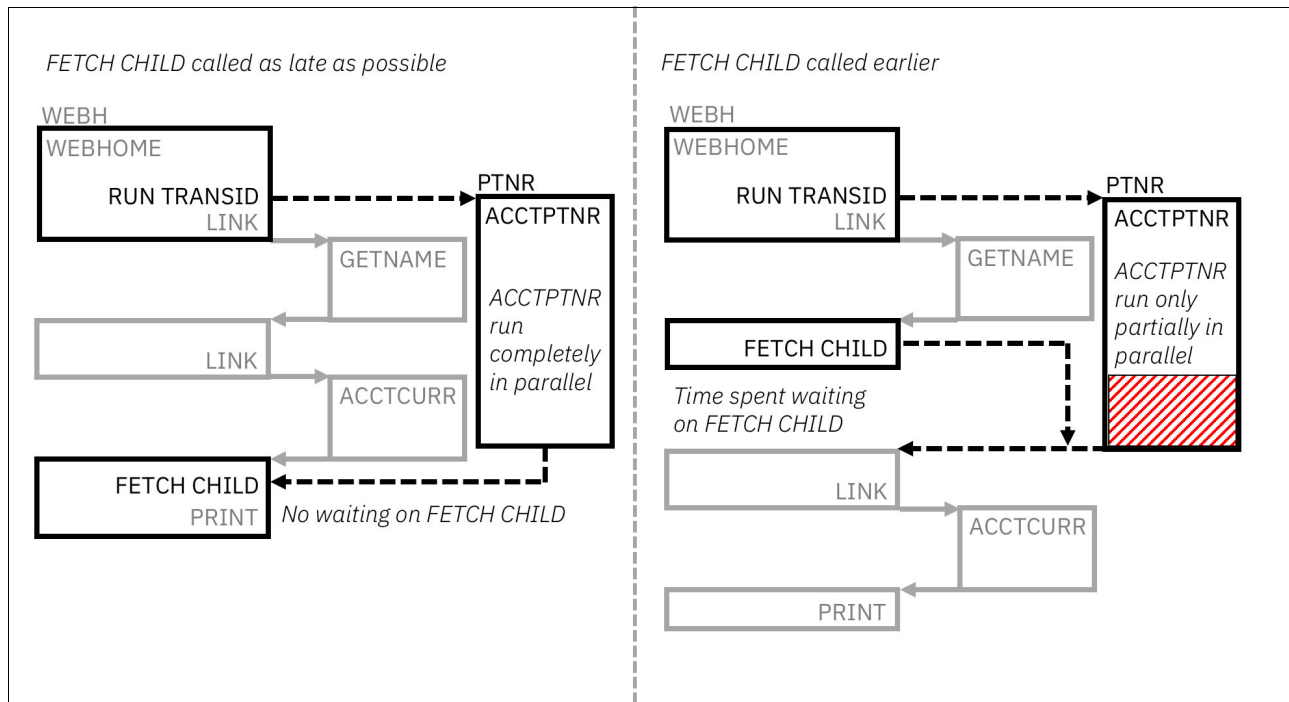


Figure 3-5 Comparing the response time savings of late versus early **FETCH CHILD** calls

**Note:** It is possible to use the **TIMEOUT** or **NOSUSPEND** options on the **FETCH CHILD** command to avoid having to wait for the child task to complete.

### Using the child token

Because the **WEBHOME** program has only one child task, you might wonder why you can't use the **FETCH ANY** command and avoid having to keep track of the **ACCTPTNR-TKN** child token that is returned from the **RUN TRANSID** call. Although this method can technically work in this case, it is a preferred practice to be specific about which child you want to fetch. If another **RUN TRANSID** call is added in the future, it will be ambiguous as to which child's results will be fetched if you use the **FETCH ANY** command and might break the **PRINT-ACCOUNT-DETAILS** code as a result of unexpected input. Specific reasons why you might use the **FETCH ANY** command are discussed further in Chapter 4, "Improving the response time of existing applications" on page 41.

### Checking the completion status of the child task

Following a **FETCH** command, check the completion status of the child task before proceeding. The **COMPSTATUS** parameter describes how the child task completed. Abends in child tasks do not flow up to parents as with the **LINK PROGRAM** command. By first checking the **COMPSTATUS** parameter, you can proceed to process the return data with confidence, knowing that the child task has completed successfully.

### Retrieving data from the child task

Finally, it is important to realize that although you used the **MYCHANNEL** channel on the **RUN TRANSID** call that started the child task, you do not use that same channel to retrieve the results. Instead, use the channel that is returned in the **CHANNEL** parameter of the **FETCH CHILD** command. This channel represents the copy of **MYCHANNEL** that was made and passed to the child task when you called the **RUN TRANSID** command. The name of the fetched channel is generated by CICS and is unique within the scope of the current link level. The fact that

channels are copied when starting child tasks avoids any issues that might arise from concurrent access.

### 3.3 Run the updated application

Now make the updated code available to CICS by using the following steps:

8. Compile the updated WEBHOME program to the data set library that is available to CICS.
9. Make the new version of the WEBHOME program available to your CICS region by issuing the following command in CICS:

```
CEMT SET PROGRAM(WEBHOME) NEW
```

10. Issue the following transaction and customer number in your CICS terminal window:

```
WEBH 0001
```

If you now run the WEBH transaction from the terminal passing the customer number 0001 you'll see the results in Example 3-9 after a few seconds.

*Example 3-9 WEBH transaction results*

---

```
TC56WEBH 20171011171430 17:14.30 Started Web banking log-on data retrieval
TC56WEBH 20171011171433 17:14.33 Welcome Pradeep Gohil
TC56WEBH 20171011171436 17:14.36 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
TC56WEBH 20171011171436 17:14.36 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
TC56WEBH 20171011171436 17:14.36 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Ended Web banking log-on data retrieval
```

---

Notice that the accounts from the partner bank are added to the printout, but the transaction didn't take any longer to complete.

### 3.4 Summary

It is often necessary to extend applications with calls to new services or requests for data. The example in this chapter shows that you can extend CICS applications without impacting their response time by using the CICS asynchronous API.

You should now feel comfortable using the **RUN TRANSID** and **FETCH CHILD** commands as an alternative to the **EXEC CICS LINK** command to start a child task to retrieve its results. You should also understand why it is important to start a child task as early as possible and fetch the results as late as possible to maximize your response time savings.

By applying what you have learned in this chapter, you are now ready to extend your applications to make them more powerful than ever, while ensuring that they remain as responsive as they always have been.

If you've read this chapter and would like to experiment further, the completed code for the scenario can be found in [Chapter 3 End Tag](#).





## Improving the response time of existing applications

This chapter walks through a scenario that shows how to use the CICS asynchronous API to improve the response time of existing applications.

In the computing world, response time is important. Markets have evolved in the digital age around value networks with responsiveness at their center. The faster you can load an app, make a stock trade, or get an answer to a search query, you are more likely to be successful at attracting users to your services over slower competition. It is increasingly common to see people abandon services, not because of a lack of capability but simply because they are “too slow.”

These market dynamics coupled with the ever-improving hardware predicted by Moore’s law<sup>1</sup>, means that engineers are challenged to hit a constantly moving target. Technology that was fast yesterday, is slow by today’s standards and will be obsolete by tomorrow. Further to the business benefit, speed is essential to meet the engineering challenges posed by a constantly growing workload that is generated from an increasingly connected world. The faster a request can be processed, the more requests can be processed in a period of time while minimizing the demand on system resources, which ultimately can reduce costs.

It is essential then, both for business and technical reasons, to constantly improve the responsiveness of critical applications to ensure they remain relevant in today’s competitive landscape. The scenario in this chapter demonstrates how to improve the response time of an existing application by using the CICS asynchronous API to optimize its existing business logic.

---

<sup>1</sup> Moore’s law, by definition, is an axiom of microprocessor development theorizing that processing power doubles about every 18 months, especially relative to cost or size.

## 4.1 Overview of the scenario

**Important process and content information:** This chapter includes a series of actions to successfully complete the described scenario. Be aware that the steps that you need to complete for this scenario are included in numbered paragraphs. Although the numbered steps might occur in different sections throughout the chapter, you still need to complete the steps in the order in which they occur.

This scenario uses the same sample application from Chapter 3, “Extending applications while minimizing the impact to response time” on page 27 to illustrate the necessary concepts. You can find the source code for the complete web banking application along with setup instructions in the `cics-async-api-redbooks` repository, under the `cicsdev` organization on [GitHub](#).

**Tip:** The repository contains the final version of the code after all the scenarios in this book are applied. If you want to follow along with this chapter, start with an earlier version of the code at this commit: [Chapter 4 Start Tag](#).

### 4.1.1 Description of the sample application

The application is written in Common Business Oriented Language (COBOL) and provides summary data for a web (mobile) banking home page (the `WEBHOME` program).

The main program, `WEBHOME.cb1`, takes a customer number as input and returns data about that customer’s accounts along with the customer’s name. The requests for the customer name and account details are performed synchronously by linking to the following programs:

- ▶ `GETNAME`
- ▶ `ACCTCURR`

The scenario described in Chapter 3, “Extending applications while minimizing the impact to response time” on page 27 added a call to the `ACCTPTNR` program to retrieve account data for accounts that the customer holds with a partner bank. This call was performed asynchronously to not impact the overall response time and to keep the application within its 9 second maximum response time, which was set by the service level agreement (SLA).

Figure 4-1 on page 43 shows the flow of the application after this addition.



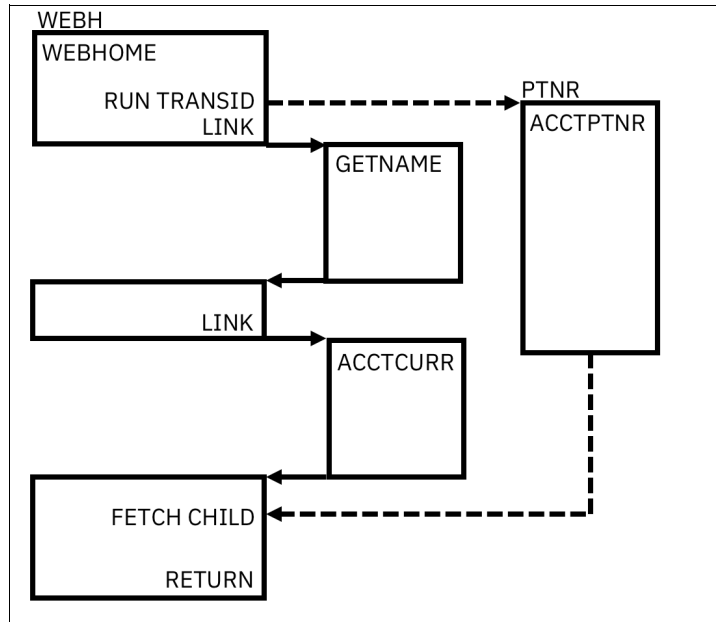


Figure 4-1 Application execution flow for WEBHOME.cbl with the addition of the ACCTPTNR program

Each of the programs that the WEBHOME program calls delays for a few seconds before returning. This delay represents the time that would have been taken for the data to be collected or calculated. The delays are greatly exaggerated from real-world timings, which likely would be subsecond. This method provides the application with a perceivable overall execution time with which to illustrate asynchronous techniques.

The specific timing of these delays is as follows:

- ▶ GETNAME: 3 seconds
- ▶ ACCTCURR: 3 seconds
- ▶ ACCTPTNR: 4 seconds

The main contributing factor to the overall response time of the application is the synchronous calls to the GETNAME and ACCTCURR programs, taking at total of 6 seconds. As the ACCTPTNR program runs asynchronously, it has no effect on the overall response time, because it takes only 4 seconds and can be performed entirely concurrently.

**Note:** When processing tasks simultaneously, it is the single, longest synchronous block that contributes to the overall execution time.

If you run the WEBH transaction from the terminal passing the 0001 customer number, you'll see the results in Example 4-1 after 6 seconds.

Issue the following transaction and customer number in your CICS terminal window, as shown in Example 4-1:

WEBH 0001

*Example 4-1 WEBH transaction results*

---

```

TC56WEBH 20171011171430 17:14.30 Started Web banking log-on data retrieval
TC56WEBH 20171011171433 17:14.33 Welcome Pradeep Gohil
TC56WEBH 20171011171436 17:14.36 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00

```

```
TC56WEBH 20171011171436 17:14.36 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
TC56WEBH 20171011171436 17:14.36 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
TC56WEBH 20171011171436 17:14.36 Ended Web banking log-on data retrieval
```

---

This example shows the name and account details for the given customer number. Notice that the program takes 6 seconds to run.

### 4.1.2 Objective of the scenario

In this scenario, you are asked to investigate ways of improving the response time of the WEBHOME program in order to provide an even more responsive experience to the web and mobile banking customers.

Typically, this task would involve looking at the business logic and data access protocols of the GETNAME, ACCTCURR, and ACCTPTNR programs to see whether you can make any optimizations to those programs. However, rewriting large chunks of the core logic in such critical programs can be risky, time consuming, and costly. Also, programs such as these likely have already been optimized many times to gain the best possible performance. Thus, any further work is likely to yield diminishing returns and provide only minor improvements to response time.

The objective of this scenario, then, is to find a way to reduce the overall response time of the application without having to modify the core business logic of the GETNAME, ACCTCURR, and ACCTPTNR programs.

## 4.2 Converting program LINKs to asynchronous API calls

The CICS asynchronous API provides a way for CICS programs to run CICS transactions and fetch the results at a later time. You can use this API to replace the LINKs to the GETNAME and ACCTCURR programs in the WEBHOME program with asynchronous **RUN TRANSID** calls. You can then use a loop combined with the **FETCH ANY** command to retrieve the results from each child in the fastest reply order.

To convert the program LINKs to asynchronous API calls:

- ▶ Define the transactions, GETN and ACUR, to run the GETNAME and ACCTCURR programs respectively.
- ▶ Add calls to the **RUN TRANSID** command at the start of the WEBHOME program to run the GETN and ACUR transactions.
- ▶ Add a **FETCH ANY** loop at the end of the WEBHOME program to collect the child task results.

By running all of the programs called by the WEBHOME program concurrently (as shown in Figure 4-2 on page 45) the response time is now dictated by the single, longest running task. In this case, that task is the ACCTPTNR program, with an execution time of 4 seconds. Thus, you can significantly reduce the overall response time by 33%, from 6 to just 4 seconds.

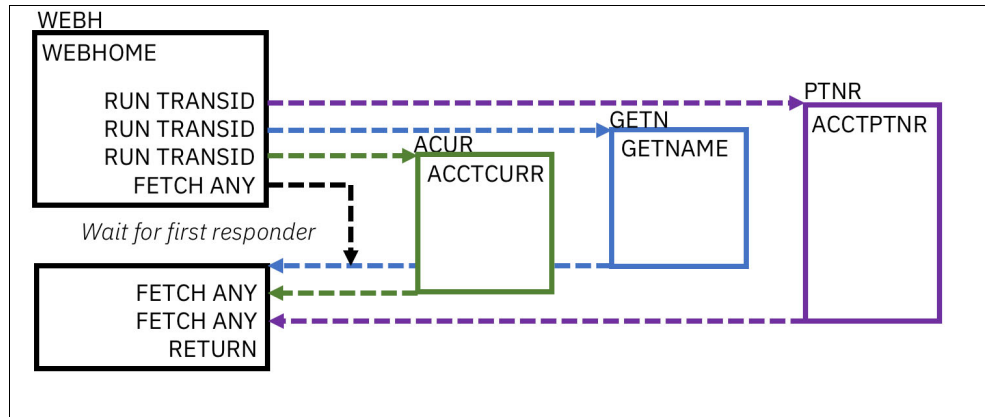


Figure 4-2 Calling all programs concurrently to reduce response time

## 4.2.1 Define transactions to run the GETNAME and ACCTCURR programs

Because the CICS asynchronous API operates on transactions not programs, you must first define transactions that you can use to run the GETNAME and ACCTCURR programs. These transactions can have any name, but for the purposes of this scenario, use GETN and ACUR.

1. Ensure that TRANSACTION definitions are defined in CICS with the attributes as shown in Example 4-2.

*Example 4-2 GETN and ACUR TRANSACTION definition attributes*

```

TRANSACTION(GETN)
GROUP(ASYNCAPI)
PROGRAM(GETNAME)
DESCRIPTION(Customer name for the CICS Asynchronous API example
)
TRANSACTION(ACUR)
GROUP(ASYNCAPI)
PROGRAM(ACCTCURR)
DESCRIPTION(Bank account details for the CICS Asynchronous API example)

```

## 4.2.2 Add RUN TRANSID commands to WEBHOME.cbl

To call the newly defined transactions from the WEBHOME program, use the **EXEC CICS RUN TRANSID** command to replace the existing **LINK** commands. This method allow you to run the GETN and ACUR transactions as child tasks of the WEBH program, meaning that they can run concurrently instead of waiting for each program to complete in turn. The next section describes the results from these child tasks.

**Tip:** If you are following this scenario using the sample on GitHub, you can find the [changes from this chapter](#) online on GitHub also.

### Adding data definitions for the GETNAME and ACCTCURR programs

You must add data field definitions for GET-NAME-TRAN, ACCTCURR-TRAN, GET-NAME-TKN and ACCTCURR-TKN to WEBHOME.cbl.

GET-NAME-TRAN and ACCTCURR-TRAN are set to the values of 'GETN' and 'ACUR' respectively. They are used to provide the transaction ID to the **RUN TRANSID** command.

GET-NAME-TKN and ACCTCURR-TKN are used to store the child tokens that you will use later to fetch the results. See 4.2.3, “Add the FETCH ANY command to WEBHOME.cbl” on page 48.

2. Add the code shown in **bold** font in Example 4-3 to the data definitions of WEBHOME.cbl.

*Example 4-3 Data definitions for GETNAME and ACCTCURR child tasks*

---

```

1 TRANSIDS.
 2 GET-NAME-TRAN PIC X(4) VALUE 'GETN'.
 2 ACCTCURR-TRAN PIC X(4) VALUE 'ACUR'.
 2 ACCTPTNR-TRAN PIC X(4) VALUE 'PTNR'.
1 CHILD-TOKENS.
 2 GET-NAME-TKN PIC X(16).
 2 ACCTCURR-TKN PIC X(16).
 2 ACCTPTNR-TKN PIC X(16).

```

---

## Replacing the LINK to the GETNAME program

3. To replace the **LINK** to the GETNAME program in WEBHOME.cbl, add the code shown in **bold** font and remove the code that is shown in ~~strike through~~ font in Example 4-4.

*Example 4-4 Replacing the LINK to the GETNAME program with the RUN TRANSID command*

---

```

* -----
* Get the customers name
* -----

EXEC CICS LINK PROGRAM (GET-NAME)

CHANNEL (MYCHANNEL)

RESP (COMMAND-RESP)

RESP2 (COMMAND-RESP2)

END-EXEC
* -----
* Asynchronously run GETN to get the customers name
* -----
EXEC CICS RUN TRANSID (GET-NAME-TRAN)

CHANNEL (MYCHANNEL)

CHILD (GET-NAME-TKN)

RESP (COMMAND-RESP)

RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

EXEC CICS GET CONTAINER (GETNAME-CONTAINER)

CHANNEL (MYCHANNEL)

INTO (CUSTOMER-NAME)

RESP (COMMAND-RESP)

RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

INITIALIZE STATUS-MSG

STRING 'Welcome '

DELIMITED BY SIZE

CUSTOMER-NAME

DELIMITED BY SIZE

INTO MSG-TEXT

PERFORM PRINT-STATUS-MESSAGE

```

```

* ----
* Get the customers current account details
* ----

```

---

## Replacing the LINK to the ACCTCURR program

4. Add the code shown in **bold** font and remove the code that is shown in ~~strikethrough~~ font in Example 4-5 to replace the **LINK** to the ACCTCURR program in WEBHOME.cb1.

*Example 4-5 Replacing the LINK to the ACCTCURR program with the RUN TRANSID command*

---

```

* ----
* Get the customers current account details
* ----

EXEC CICS LINK PROGRAM (ACCTCURR)
CHANNEL (MYCHANNEL)
RESP (COMMAND-RESP)
RESP2 (COMMAND-RESP2)
END-EXEC

* -----
* Asynchronously run ACUR to get customers
* current account details
* -----
EXEC CICS RUN TRANSID (ACCTCURR-TRAN)
CHANNEL (MYCHANNEL)
CHILD (ACCTCURR-TKN)
RESP (COMMAND-RESP)
RESP2 (COMMAND-RESP2)
END-EXEC

PERFORM CHECK-COMMAND

EXEC CICS GET CONTAINER (ACCTCURR-CONTAINER)
CHANNEL (MYCHANNEL)
INTO (CURRENT-ACCOUNTS)
RESP (COMMAND-RESP)
RESP2 (COMMAND-RESP2)
END-EXEC
PERFORM CHECK-COMMAND
PERFORM PRINT-CURRENT-ACCOUNTS-DETAILS

* -----
* Get the customers current account details from the
* partner bank
* -----

```

---

**Note:** Both Example 4-4 and Example 4-5 do not just removed the **LINK** commands but also remove the logic that was used to process the results from the GETNAME and ACCTCURR programs. You will add back this logic later in 4.2.3, “Add the FETCH ANY command to WEBHOME.cb1” on page 48 when you fetch the results from the child tasks.

Keep in mind the following important concepts at this stage:

- Passing data to programs that were previously LINKed to

- Removing the results processing logic

### Passing data to programs that were previously LINKed to

Because the GETNAME and ACCTCURR programs have a channel and container-based interface, the transition to call them asynchronously is relatively smooth. You can simply pass the same channel that you previously used on the **LINK** command to the **RUN TRANSID** command. If these programs had COMMAREA interfaces, you would need a different process, because the CICS asynchronous API is based on passing channels to child tasks and uses these channels to manage the results.

Calling COMMAREA-based applications using the CICS asynchronous API requires a stub program. There are more details about how to do this in 7.2, “Tip: Run existing COMMAREA-based assets asynchronously without changing them” on page 106.

### Removing the results processing logic

When replacing **LINK** commands with asynchronous calls, it is important to also remove any logic following the link that acted on the results. Because control is now being returned to the parent immediately, rather than waiting for the program to complete, the results are not yet available to be acted upon. Move results processing logic to the point where the child task's results are fetched, as described in 4.2.3, “Add the **FETCH ANY** command to **WEBHOME.cbl**” on page 48.

## 4.2.3 Add the **FETCH ANY** command to **WEBHOME.cbl**

Now that you are running multiple child tasks, the next step is to retrieve their results. You could call **FETCH CHILD** multiple times, passing each child token in turn. However this method will not maximize the response time savings. Ideally, you want to fetch the results in any order in which the child tasks complete. Taking this approach allows the parent task to process results as soon as they become available and minimizes the time spent waiting.

If you were to fetch each child specifically, there is no way to know that you are fetching the results in the optimal order. You can make a best guess, but the dynamic nature of a running system means that it's likely you will be wrong at least some of the time when requests execute slower or faster than expected. Figure 4-3 illustrates the missed response time savings from not fetching the first responder first.

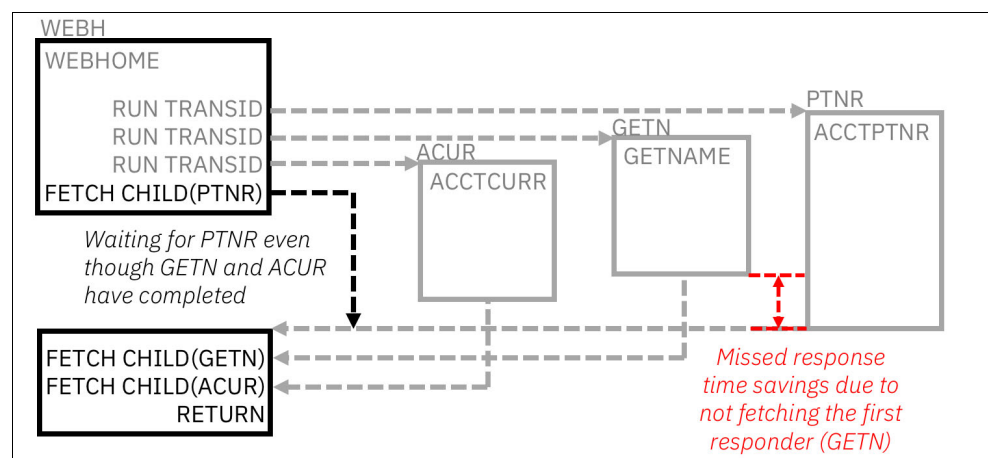


Figure 4-3 Missed response time savings from not fetching the first responder

The CICS asynchronous API provides a solution to this problem in the form of the **FETCH ANY** command. You can use this command in combination with a loop to fetch the results from child tasks in the most efficient manner. By using **FETCH ANY** you can ensure that you are always processing results in the most optimal order and maximizing response time savings.

## Adding data definitions

You need to add the following data definitions for some fields that aid in the fetching and processing of the child tasks' results:

- ▶ ANY-CHILD-TKN is used to store the child token returned on the **FETCH ANY** command.
- ▶ ANY-CHILD-CHAN is used to store the reply channel name returned on the **FETCH ANY** command.
- ▶ GET-NAME-CHAN is used to store the reply channel name for the GETN child task.
- ▶ ACCTCURR-CHAN is used to store the reply channel name for the ACUR child task.

5. Add the code shown in **bold** font in Example 4-6 to the data definitions of WEBHOME.cbl.

*Example 4-6 Data definitions for the FETCH ANY loop*

---

```

1 CHILD-TOKENS.
 2 ANY-CHILD-TKN PIC X(16).
 2 GET-NAME-TKN PIC X(16).
 2 ACCTCURR-TKN PIC X(16).
 2 ACCTPTNR-TKN PIC X(16).
1 RETURN-CHANNELS.
 2 ANY-CHILD-CHAN PIC X(16).
 2 GET-NAME-CHAN PIC X(16).
 2 ACCTCURR-CHAN PIC X(16).
 2 ACCTPTNR-CHAN PIC X(16).
```

---

## Replacing FETCH CHILD with FETCH ANY

Next, you need to replace the existing singular **FETCH CHILD** command with a **FETCH ANY** loop.

6. In WEBHOME.cbl, add the code shown in **bold** font and remove the code that is shown in ~~strikethrough~~ font in Example 4-7.

*Example 4-7 Replacing FETCH CHILD with FETCH ANY*

---

```

* -----
* Get the customers current account details from the
* partner bank
* -----
EXEC CICS FETCH CHILD (ACCTPTNR-TKN)
 CHANNEL (ACCTPTNR-CHAN)
 COMPSTATUS (CHILD-RETURN-STATUS)
 ABCODE (CHILD-RETURN-ABCODE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)
END-EXEC
PERFORM CHECK-COMMAND
PERFORM CHECK-CHILD
EXEC CICS GET-CONTAINER (ACCTPTNR-CONTAINER)
 CHANNEL (ACCTPTNR-CHAN)
 INTO (PARTNER-ACCOUNTS)
 RESP (COMMAND-RESP)
```

---

```

_____RESP2_____ (COMMAND-RESP2)
_____END-EXEC
_____PERFORM CHECK-COMMAND
_____PERFORM PRINT-PARTNER-ACCOUNTS-DETAILS

```

```

* -----
* Three child tasks have been run to execute asynchronously.
* Loop through the children to get the customer's details
* -----

```

PERFORM 3 TIMES

```

EXEC CICS FETCH ANY (ANY-CHILD-TKN)
 CHANNEL (ANY-CHILD-CHAN)
 COMPSTATUS (CHILD-RETURN-STATUS)
 ABCODE (CHILD-RETURN-ABCODE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

```

END-EXEC

PERFORM CHECK-COMMAND

PERFORM CHECK-CHILD

```

* -----
* Identify which child completed and process results
* -----
EVALUATE ANY-CHILD-TKN

```

```

* -----
* For GETNAME, print the welcome message

```

```

* -----
WHEN GET-NAME-TKN

```

```

* -----
* For ACCTCURR, print the account details
* -----
WHEN ACCTCURR-TKN

```

```

* -----
* For ACCTPTNR, print the partner account details
* -----
WHEN ACCTPTNR-TKN

```

```

* -----
* Error: Unknown child is returned
* -----

```

WHEN OTHER

```

INITIALIZE STATUS-MSG
STRING '*** Unknown child token: '
 DELIMITED BY SIZE
 ANY-CHILD-TKN
 DELIMITED BY SIZE
 INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

```

PERFORM WEBHOME-ERROR



## END-EVALUATE

\* End of FETCH ANY loop  
END-PERFORM

\* Send a message to the screen to  
\* notify terminal user of completion  
MOVE 'COMPLETE' TO CURRENT-STATUS  
PERFORM PRINT-TEXT-TO-SCREEN

---

This is a large chunk of code, so let's break it down from the top.

First, there is a simple loop, controlled by the PERFORM 3 TIMES statement. In this scenario there are always three child tasks, so this statement is the easiest way to control the number of times that you execute the loop.

**Tip:** If the number of child tasks is variable or unknown, inspect the RESP code on the **FETCH ANY** command, and use the NOTFND return code to control when to exit the loop.

Next, the **FETCH ANY** command fetches the next completed child task that has not yet been fetched. If all the unfetched child tasks are still running, it waits until one completes. The code then stores the token for this child in ANY-CHILD-TKN and the channel name in ANY-CHILD-CHAN.

Finally, the code uses an EVALUATE statement to inspect the ANY-CHILD-TKN to see which of the child tasks have been fetched. The three tokens returned by the earlier **RUN TRANSID** calls are used in the WHEN clauses to match against. If a child task is fetched with a token other than these three then it is treated as an error, because it probably means that someone added a new child task without updating this loop.

The results processing logic for each child task is left out intentionally at this stage to help keep the structure of this loop clear.

### Adding results processing logic to the EVALUATE statement

With the skeleton EVALUATE statement in place, the final task is to add logic to each of the WHEN clauses to process the results for each child.

7. Add the code shown in **bold** font in Example 4-8 to the EVALUATE statement.

*Example 4-8 Adding results processing logic to our EVALUATE statement*

---

```
* -----
* Identify which child completed and process results
* -----
* EVALUATE ANY-CHILD-TKN
* -----
* For GETNAME, print the welcome message
* -----
* WHEN GET-NAME-TKN

* Save the channel name for future use
* MOVE ANY-CHILD-CHAN TO GET-NAME-CHAN
* EXEC CICS GET CONTAINER (GETNAME-CONTAINER)
* CHANNEL (GET-NAME-CHAN)
* INTO (CUSTOMER-NAME)
```

```

 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)
END-EXEC
PERFORM CHECK-COMMAND
INITIALIZE STATUS-MSG
STRING 'Welcome '
 DELIMITED BY SIZE
 CUSTOMER-NAME
 DELIMITED BY SIZE
 INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

* -----
* For ACCTCURR, print the account details
* -----
 WHEN ACCTCURR-TKN

* Save the channel name for future use
 MOVE ANY-CHILD-CHAN TO ACCTCURR-CHAN
 EXEC CICS GET CONTAINER (ACCTCURR-CONTAINER)
 CHANNEL (ACCTCURR-CHAN)
 INTO (CURRENT-ACCOUNTS)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

 END-EXEC

 PERFORM CHECK-COMMAND
 PERFORM PRINT-CURRENT-ACCOUNTS-DETAILS

* -----
* For ACCTPTNR, print the partner account details
* -----
 WHEN ACCTPTNR-TKN

* Save the channel name for future use
 MOVE ANY-CHILD-CHAN TO ACCTPTNR-CHAN

 EXEC CICS GET CONTAINER (ACCTPTNR-CONTAINER)
 CHANNEL (ACCTPTNR-CHAN)
 INTO (PARTNER-ACCOUNTS)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

 END-EXEC

 PERFORM CHECK-COMMAND
 PERFORM PRINT-PARTNER-ACCOUNTS-DETAILS

* -----
* Error: Unknown child is returned
* -----

```

---

You'll notice that the results processing logic is nearly identical to the logic that you removed earlier in this chapter. This logic highlights how easy it is to transition to running programs asynchronously rather than using **LINKs**, because you are moving, rather than rewriting, existing logic.

**Note:** In this scenario, the programs **LINKed** to are not dependant on each other, meaning that the output of one is not used as the input to another. This method makes the programs excellent candidates for asynchronous processing, and it is relatively easy to do.

In a situation where you have programs that depend on each other's output, you can still take advantage of asynchronous processing. Run your child tasks in blocks where the results from tasks in one block are used as input to the tasks in the next. For more information about running collections of child tasks, see 7.4, "Trick: Prevent sets of children from interfering in FETCH ANY logic by using FREE CHILD" on page 117.

### Saving the return channel

The main change to the logic is that you must use the channel name that is returned on the **FETCH ANY** command to get containers. You can use ANY-CHILD-CHAN in each of the GET CONTAINER calls; however, this field is overwritten on each iteration.

It is preferred practice to copy this channel name to a field that is specific to each child for any future uses. For example:

```
MOVE ANY-CHILD-CHAN TO GET-NAME-CHAN
```

Using the CHANNEL parameter on a **FETCH** command transfers the child's channel to the parent and generates a new unique channel name. This action can be issued only once per child; therefore, save it in case you need it at any point later in your business logic.

## 4.3 Running the updated application

Compile the sample and NEWCOPY your programs in CICS. If you now run the WEBH transaction from the terminal passing the customer number 0001, you'll see the results in Example 4-9 after a few seconds.

*Example 4-9 WEBH transaction results*

---

|          |                |          |                                           |                |                     |
|----------|----------------|----------|-------------------------------------------|----------------|---------------------|
| T142WEBH | 20171012174409 | 17:44.09 | Started Web banking log-on data retrieval |                |                     |
| T142WEBH | 20171012174412 | 17:44.12 | Welcome Pradeep Gohil                     |                |                     |
| T142WEBH | 20171012174412 | 17:44.12 | Acc: 20140720                             | Bal: £0.01     | Overdraft: £0.00    |
| T142WEBH | 20171012174412 | 17:44.12 | Acc: 25875343                             | Bal: £45742.00 | Overdraft: £1000.00 |
| T142WEBH | 20171012174412 | 17:44.12 | Acc: 20170125                             | Bal: £34533.23 | Overdraft: £0.00    |
| T142WEBH | 20171012174413 | 17:44.13 | Acc: 62837456                             | Bal: £234.56   | Overdraft: £0.00    |
| T142WEBH | 20171012174413 | 17:44.13 | Acc: 64620987                             | Bal: £3092.60  | Overdraft: £1000.00 |
| T142WEBH | 20171012174413 | 17:44.13 | Acc: 64563923                             | Bal: £10123.98 | Overdraft: £0.00    |
| T142WEBH | 20171012174413 | 17:44.13 | Ended Web banking log-on data retrieval   |                |                     |

---

Notice that the program performs the same work as before, but now takes only 4 seconds to run (when it previously took 6 seconds). With the GETNAME and ACCTCURR programs now running asynchronously, the overall response time is dictated by the ACPTPTNR program.

## 4.4 Summary

Often, you need to improve the response time of applications either to make room for more functions or simply to keep pace with the demands of users. The example in this chapter shows how to improve the response time of CICS applications by using the CICS asynchronous API to call programs asynchronously.

After following the example in this chapter, you should feel comfortable using the **RUN TRANSID** and **FETCH ANY** command, as an alternative to **EXEC CICS LINK** commands, to start multiple child tasks and retrieve their results. You should also understand how the **FETCH ANY** command allows you to fetch child tasks as soon as they complete and appreciate that doing so allows you to maximize response time savings.

By applying the methods in this chapter you can optimize applications to make them more responsive than ever while keeping most of their existing logic intact.

If you've read this chapter and would like to experiment further, the completed code for the scenario can be found in [Chapter 4 End Tag](#).



## Developing robust applications with unreliable service providers

This chapter demonstrates how to use the CICS asynchronous API to create robust CICS applications that protect against unreliable service calls while also maintaining an expected response time goal.

As discussed in previous chapters, CICS applications are evolving and harnessing services that are offered by other providers. Using third-party services can introduce an element of risk because service providers are subject to their own system behaviors. A spike in requests, hardware issues, and poor design can all affect a provider's ability to respond reliably. When things do go wrong or response times degrade, the false impression is that the CICS application is at fault, when often the culprit is a service provider.

Rather than losing new business opportunities, the scenario in this chapter provides an example where the `TIMEOUT` feature on **`FETCH`** commands ensures that CICS applications maintain their robust and responsive qualities of service.

## 5.1 Overview of the scenario

**Important process and content information:** This chapter includes a series of actions to successfully complete the described scenario. Be aware that the steps that you need to complete for this scenario are included in numbered paragraphs. Although the numbered steps might occur in different sections throughout the chapter, you still need to complete the steps in the order in which they occur.

This scenario builds upon the fictional web banking home page example, developed over the previous chapters. You can find the source code for the complete web banking application along with setup instructions in the `cics-async-api-redbooks` repository, under the `cicsdev` organization on [GitHub](#).

**Tip:** The repository contains the final version of the code after all of the scenarios in this book are applied. If you want to follow the scenario described in this chapter, start with an earlier version of the code at this commit: [Chapter 5 Start Tag](#).

The example banking application is enhanced to use a new business opportunity. To encourage the upsell of personal loans, a personalized loan quote is offered as part of the banking home page. However, with the existing business logic and service calls in the banking home page application, the addition of the loan quote threatens the response time goals of the CICS application.

The scenario in this chapter looks at how a new service can be added while maintaining the application's current service level agreement (SLA). The new service provider for loan quotes has an unpredictable response time. This scenario enhances the example to provide a robust application, by using timeouts, that can protect against unreliable services.

### 5.1.1 Objective of the scenario

The business intention for this scenario is to upsell the companies range of personal loans by providing a personalized quote, as part of the WEBHOME program. The web banking home page program, WEBHOME, currently retrieves customer details, from the GETNAME program, along with current accounts that are held with the bank and banking partner, in the ACCTCURR and ACCTPTNR program, to populate the initial screen when a customer logs on to the Internet banking page.

This core customer-facing banking application has a response time goal of 9 seconds. This goal might seem to be an excessive response goal; however, the times are exaggerated in this example for illustrative purposes. Due to the use of the CICS asynchronous API in prior chapters, the current response time of the WEBHOME program is good and affords enhancements for new business opportunities.

The service to provide a loan quote, GETLOAN, is expected to take the application to its response time goal limit. The WEBHOME program can use the loan quote if it replies in time (achieved by setting a timeout on the GETLOAN call). Otherwise, it needs to abandon the quote to maintain its SLA, thereby providing a robust CICS application that is protected against the unreliable service.

## 5.2 Requesting services from an unreliable service provider

The WEBHOME program includes a new feature that provides a loan quote to the customer as part of the initial web banking home page. Unlike competitors that bombard customers with products that are not suited to them, this feature provides a personalized quote based on the customer's current banking products.

The existing business logic in the WEBHOME program retrieves bank account details held with the bank and partner banking provider. These results are provided to the GETLOAN loan quote service to provide a personalized rate.

Due to the personalization of the new quoting feature, it cannot be run immediately at the start of the WEBHOME program (as with the other service requests in the program). The GETLOAN service must be requested after the return of the ACCTCURR and ACCTPTNR programs. The results of the ACCTCURR and ACCTPTNR programs are first written to a new container for input to the GETLOAN service, and then the GETLOAN service can be requested.

Figure 5-1 shows the order of all service requests made in the WEBHOME program. The GETNAME service and the ACCTCURR and ACCTPTNR programs all run concurrently, and the results are processed as the child tasks complete. The GETLOAN service runs and is fetched after the other service requests.

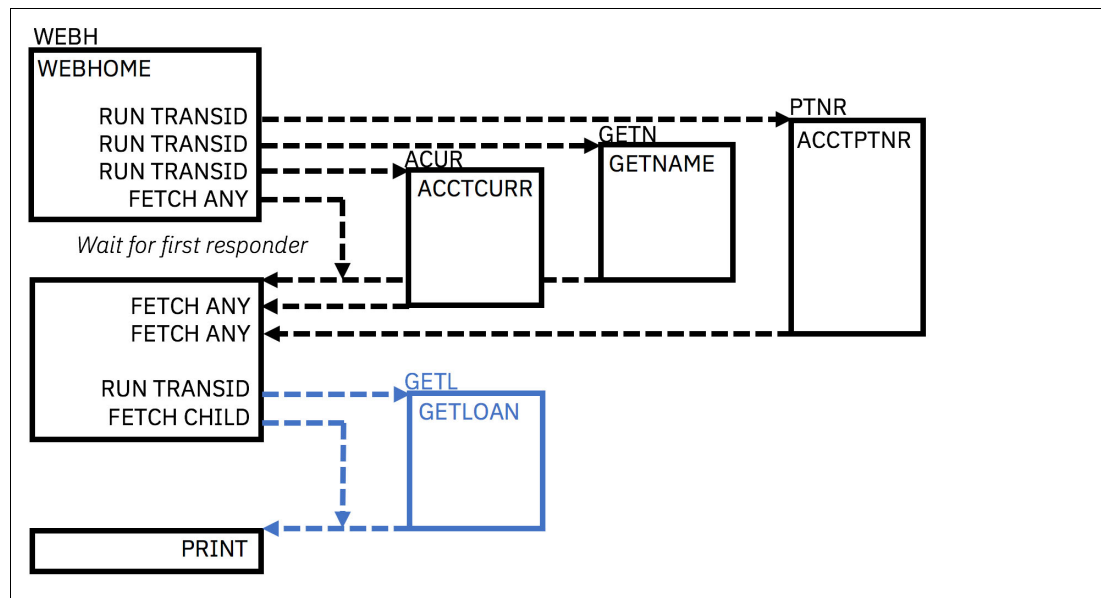


Figure 5-1 Order of child service requests made in the WEBHOME program

**Note:** You can further optimize the business logic in the WEBHOME program for the GETLOAN service to run concurrently with the GETNAME program. Thus, running the GETLOAN child task can occur during the current **FETCH ANY** looping algorithm. However, for simplicity, this scenario runs and fetches the GETLOAN service after the existing service requests.

In addition, the GETNAME service normally returns first, so complicating the **FETCH ANY** loop is not beneficial.

## 5.2.1 Why not just use a LINK?

As shown in Figure 5-1 on page 57, the request and consumption to the GETLOAN service occurs at the end of the WEBHOME program. It is not intended to run concurrently with any other requests. A reasonable solution to the problem (at this stage in the design) is to link to the GETLOAN service. By issuing an **EXEC CICS LINK PROGRAM(GETLOAN)** command, control can be passed to the GETLOAN services, as shown in Figure 5-2.

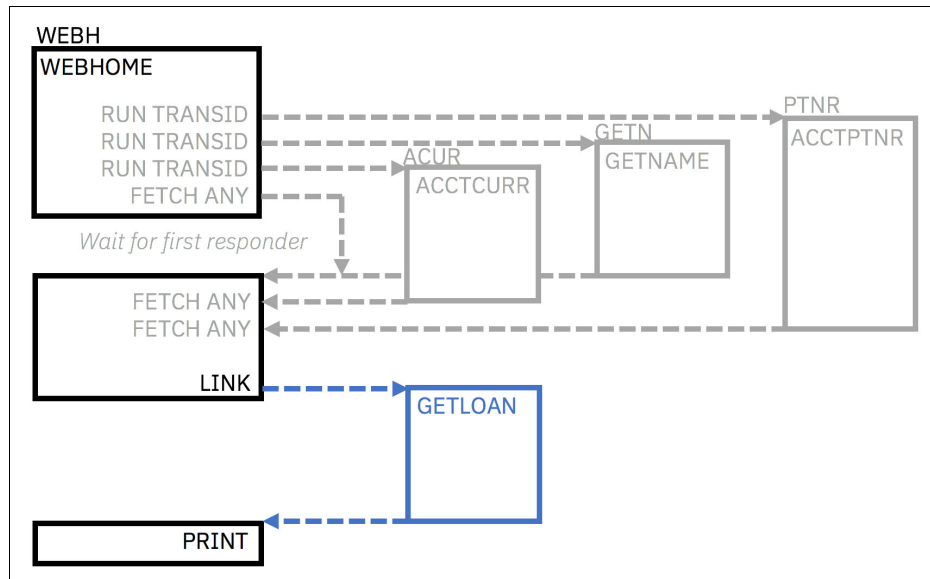


Figure 5-2 Using a LINK to initiate the GETLOAN service

As expected, on the **LINK** command, you can specify a channel to pass data to the GETLOAN service. As soon as the GETLOAN service completes and returns, control is passed back to the WEBHOME program, and the channel is updated with the result set from the GETLOAN service. Using a **LINK** rather than asynchronously running a child provides the following benefits:

- ▶ There are fewer commands to issue (one **LINK** versus a **RUN TRANSID** and **FETCH CHILD** command).
- ▶ It does not require an additional transaction to run the GETLOAN service.
- ▶ There is less cognitive load on the application developer, because the business logic transfers over to the GETLOAN service.

However, using the CICS asynchronous API to request the GETLOAN service provides the following benefits:

- ▶ Maintaining control in the parent WEBHOME program. (With a **LINK**, execution control is handed to the next program, and the caller does not regain control until the call is completed.)
- ▶ There is no timeout on a **LINK** command.
- ▶ Future proofing the application, as explained in the paragraph that follows.

The design architect for the WEBHOME program understands that if this venture becomes successful, other products in the company's portfolio are likely to follow a similar pattern, such as upselling of savings accounts, mortgages, insurance policies, and so on. If future services are called sequentially, the response time of the WEBHOME program will degrade. To encourage asynchronous behaviour, it is decided to continue with the existing calling pattern in the WEBHOME program and call the GETLOAN service asynchronously.



This example demonstrates that the CICS asynchronous API can provide additional benefits to application logic when a single service is requested. The asynchronous API is not limited to scenarios with multiple child tasks.

## 5.2.2 Asynchronously requesting a new service

To add the request for the personalized loan quote:

- ▶ Define a GETL transaction, which invokes the GETLOAN loan quote service.
- ▶ Make available a container called ALLCUSTACCOUNTS, which provides the input to a personalized loan quote service.
- ▶ Asynchronously run the GETLOAN service by initiating the GETL transaction, fetch the personalized loan quote, and display the loan quote result in a status message.

The loan rate service also requires the customer account number. However, this number is already stored in a container, so no additional work is required to provide this information.

### Define the GETL transaction to run the GETLOAN program

Because the CICS asynchronous API operates on transactions not programs, you need to first define a transaction that you can use to run the GETLOAN service. This transaction can have any name, but for the purposes of this scenario, use GETL.

1. Ensure that a TRANSACTION definition is defined in CICS with the attributes shown in Example 5-1.

*Example 5-1 GETL TRANSACTION definition attributes*

---

```
TRANSACTION(GETL)
GROUP(ASYNCAPI)
PROGRAM(GETLOAN)
DESCRIPTION(Transaction to request a personalized loan rate for the CICS
Asynchronous API Redbooks sample)
```

---

### Declare the container constants, transaction identifiers, and token record structures

2. Add the lines shown in **bold** font in Example 5-2 to the WEBHOME program to declare the container names.

*Example 5-2 Adding variables for calling the personalized loan quote service*

---

```
1 CONTAINER-NAMES.
2 INPUT-CONTAINER PIC X(16) VALUE 'INPUTCONTAINER '.
2 GETNAME-CONTAINER PIC X(16) VALUE 'GETNAMECONTAINER'.
2 ACCTCURR-CONTAINER PIC X(16) VALUE 'ACCTCURRCONT '.
2 ACCTPTNR-CONTAINER PIC X(16) VALUE 'ACCTPTNRCONT '.
2 GETLOAN-CONTAINER PIC X(16) VALUE 'GETLOANCONTAINER'.
2 ACCOUNTS-CONTAINER PIC X(16) VALUE 'ALLCUSTACCOUNTS '.
```

---

The ACCOUNT-CONTAINER contains all the customer accounts that are retrieved by the ACCTCURR and ACCTPTNR programs. This information is passed as input to the GETLOAN service. The GETLOAN-CONTAINER contains the results of the personalized quote following the return of the GETLOAN service.

3. Add the lines shown in **bold** in Example 5-3 to the WEBHOME program to declare the transaction identifier and token variables.

*Example 5-3 Adding the transaction ID and variables for the GETLOAN child*

---

```

1 TRANSIDS.
2 GET-NAME-TRAN PIC X(4) VALUE 'GETN'.
2 ACCTCURR-TRAN PIC X(4) VALUE 'ACUR'.
2 ACCTPTNR-TRAN PIC X(4) VALUE 'PTNR'.
2 GETLOAN-TRAN PIC X(4) VALUE 'GETL'.
1 CHILD-TOKENS.
2 ANY-CHILD-TKN PIC X(16).
2 GET-NAME-TKN PIC X(16).
2 ACCTCURR-TKN PIC X(16).
2 ACCTPTNR-TKN PIC X(16).
2 GET-LOAN-TKN PIC X(16).
1 RETURN-CHANNELS.
2 ANY-CHILD-CHAN PIC X(16).
2 GET-NAME-CHAN PIC X(16).
2 ACCTCURR-CHAN PIC X(16).
2 ACCTPTNR-CHAN PIC X(16).
2 GET-LOAN-CHAN PIC X(16).

```

---

The GETLOAN-TRAN defines the GETL transaction, which initiates the GETLOAN service. The GET-LOAN-TKN is used to track the child token that is returned on the **RUN TRANSID** command and is used later on the **FETCH CHILD** command. The GET-LOAN-CHAN is used to store the returned channel name on the fetch of the results.

## Prepare, run, fetch, and display the personalized loan quote service

After the existing **FETCH ANY** logic in the WEBHOME program, put a new container on the channel and run a child task to request the GETLOAN service. At this stage, then immediately fetch the results from the child. This process has the effect of blocking the parent execution during the **FETCH CHILD** command until the GETLOAN child completes.

The final step is to print the results to the log messages to determine the loan rate.

4. Add to the code (shown in **bold** font in Example 5-4), to put the customer's account details into a new container, and to run the GETLOAN child task. The logic then fetches the results and prints them as a status message.

*Example 5-4 Prepare the input data, and call the GETLOAN service asynchronously*

---

```

* End of FETCH ANY loop
 END-PERFORM

* -----
* Provide new business directive of Loan up-sell.
* Asynchronously call personalized loan rate generator.
* -----

* -----
* Pass the details of all of the customer's accounts

```

---

```

* to provide a personalized loan quote
* -----
EXEC CICS PUT CONTAINER (ACCOUNTS-CONTAINER)
 FROM (CUSTOMER-ACCOUNTS)
 CHANNEL (MYCHANNEL)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

* -----
* Asynchronously run GETL to get customers
* personalized loan rate
* -----
EXEC CICS RUN TRANSID (GETLOAN-TRAN)
 CHANNEL (MYCHANNEL)
 CHILD (GET-LOAN-TKN)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

* -----
* Perform the FETCH of loan rate
* -----
EXEC CICS FETCH CHILD (GET-LOAN-TKN)
 CHANNEL (GET-LOAN-CHAN)
 COMPSTATUS (CHILD-RETURN-STATUS)
 ABCODE (CHILD-RETURN-ABCODE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND
PERFORM CHECK-CHILD

* -----
* Successful response from the child.
* Get the personalized loan quote
* -----
EXEC CICS GET CONTAINER (GETLOAN-CONTAINER)
 CHANNEL (GET-LOAN-CHAN)
 INTO (CUSTOMER-LOAN-RATE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

* -----
* Finally, display the loan quote
* -----

```

```

INITIALIZE STATUS-MSG
STRING 'Personalized Loan Rate: '
 DELIMITED BY SIZE
 CUSTOMER-LOAN-RATE
 DELIMITED BY SPACE
 ' %'
 DELIMITED BY SIZE
 INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

```

```

* Send a message to the screen to
* notify terminal user of completion
MOVE 'COMPLETE' TO CURRENT-STATUS
PERFORM PRINT-TEXT-TO-SCREEN

```

---

The first stage of requesting the GETLOAN child from the WEBHOME program is now complete. As part of the initial example set up, you already have the GETLOAN service available in the CICS region.

5. Compile the updated WEBHOME program to the data set library that is available to CICS.
6. Make the new version of the WEBHOME program available to your CICS region by issuing the following command in CICS:

```
CEMT SET PROGRAM(WEBHOME) NEW
```

### 5.2.3 Testing the response times of calling the new service

After deploying the modified WEBHOME program with the updates (as described in 5.2.2, “Asynchronously requesting a new service” on page 59), you can test the program by issuing the WEBH transaction, along with a customer number, such as 0001.

7. Issue the following transaction and customer number in your CICS terminal window:

```
WEBH 0001
```

The updated WEBHOME program executes and runs a child task to obtain a personalized loan quote, which is then fetched and displays as a status message in the CICS language environment messages log, such as CEEMSG. Example 5-5 shows the results of such an execution of the WEBHOME program. Important parts of the status messages are highlighted in **bold font**.

*Example 5-5 Status messages written from the web banking example to CEEMSG*

---

```

T127WEBH 20171008020827 02:08.27 Started Web banking log-on data retrieval
T127WEBH 20171008020830 02:08.30 Welcome Pradeep Gohil
T127WEBH 20171008020830 02:08.30 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
T127WEBH 20171008020830 02:08.30 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
T127WEBH 20171008020830 02:08.30 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
T127WEBH 20171008020831 02:08.31 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
T127WEBH 20171008020831 02:08.31 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
T127WEBH 20171008020831 02:08.31 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
 GETL 20171008020831 Loan quote service under normal load. ETA 4 secs.
T127WEBH 20171008020835 02:08.35 Personalized Loan Rate: 1.25 %
T127WEBH 20171008020835 02:08.35 Ended Web banking log-on data retrieval

```

---

The status messages written by the banking example show that there is an 8-second interval between the start and end messages. This delay is within the 9-second SLA response goal for the web banking home application.

You can also see from the messages that the GETLOAN service (the GETL transaction) is running under a normal load. It can respond to the caller in 4 seconds. The WEBHOME program can obtain and display the customer personalized loan rate, and in this example it is 1.25%.

If you re-test the web banking example multiple times, by reissuing **WEBH 0001**, notice that on some occasions the response time of the GETLOAN service can vary. Example 5-6 shows the messages when the loan quote service is under a heavy load.

*Example 5-6 Web banking example status messages when loan service under heavy load*

---

```
T127WEBH 20171008020907 02:09.07 Started Web banking log-on data retrieval
T127WEBH 20171008020910 02:09.10 Welcome Pradeep Gohil
T127WEBH 20171008020910 02:09.10 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
T127WEBH 20171008020910 02:09.10 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
T127WEBH 20171008020910 02:09.10 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
T127WEBH 20171008020911 02:09.11 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
T127WEBH 20171008020911 02:09.11 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
T127WEBH 20171008020911 02:09.11 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
GETL 20171008020911 Loan quote service under heavy load. ETA 7 secs.
T127WEBH 20171008020918 02:09.18 Personalized Loan Rate: 1.25 %
T127WEBH 20171008020918 02:09.18 Ended Web banking log-on data retrieval
```

---

Example 5-6 shows that under heavy load the response time of the loan service changes from 4 to 7 seconds, which has the undesired effect of reducing the response time of the application to 11 seconds and exceeds the 9-second goal.

Figure 5-3 on page 64 depicts the effect of the unreliable GETLOAN service against the response time goals of the WEBHOME program. The left diagram in the figure shows that when the GETLOAN service replies in a timely manner, the WEBHOME program can achieve its SLA. However, because the **FETCH CHILD** command suspends the parent task until the GETLOAN service replies, the right diagram in the figure shows that the WEBHOME program misses its response time SLA.

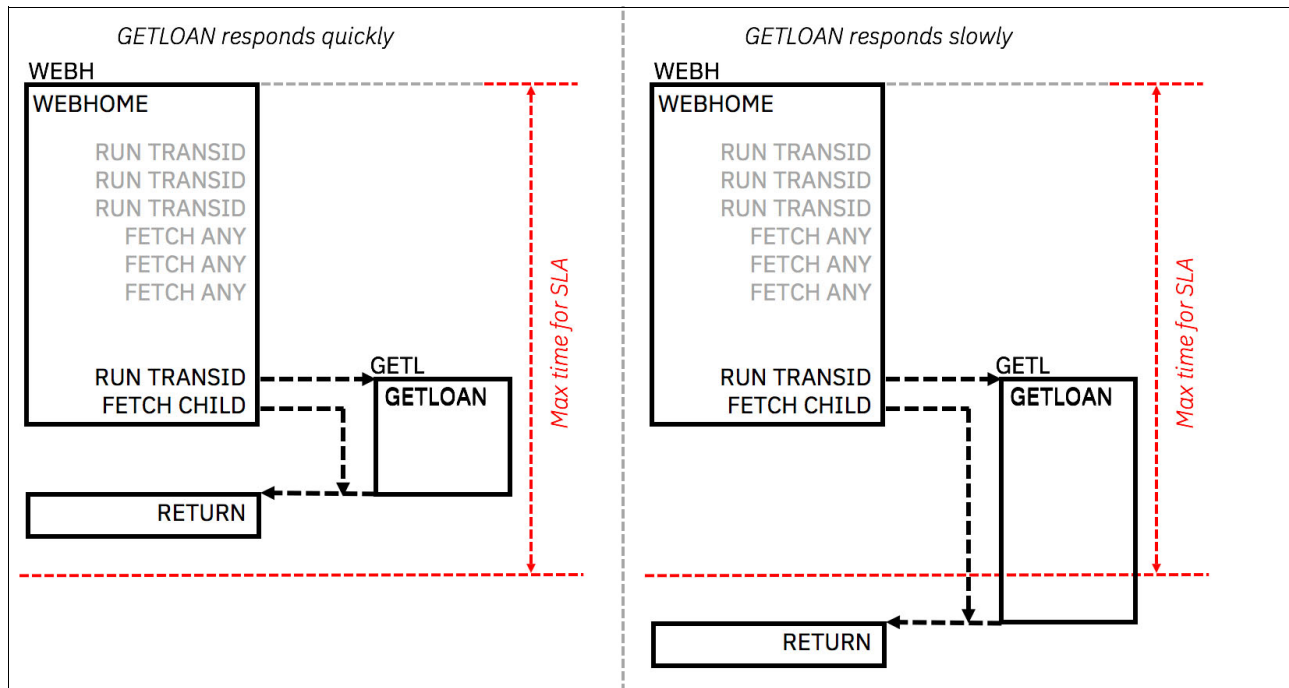


Figure 5-3 The effect the GETLOAN service has on the response time goals of the WEBHOME program

The agreed future approach is for the personalized quote to be used only when within the agreed SLA. In situations where the response of the GETLOAN service takes too long, the process should be abandoned in favour of achieving the SLA.

Use a timeout value on the **FETCH CHILD** command to control how long the WEBHOME program is willing to wait for a response. The GETNAME, ACCTCURR, and ACCTPTNR child tasks usually take approximately 4 seconds to complete. This wait time affords 5 seconds to the GETLOAN service to keep within the SLA.

**Note:** In practice, it is likely that a similar scenario would use a current time function, along with the starting time of the application, to calculate the time left for completing child tasks. However, for simplicity of showcasing the timeout feature of the CICS asynchronous API, this scenario uses a 5-second timeout.

## 5.2.4 Retrieving a timeout value to meet the application's SLA

This example adds a 5-second timeout on the **FETCH CHILD** command for the GETLOAN service request. You can use the follow methods to achieve this timeout value:

- ▶ Set the timeout value explicitly in the application code.
- ▶ Parameterize the timeout value.

If the timeout value is obvious and will not change, it is trivial to *hardcode* the value in application code. However, it is often the case that a predetermined timeout value might not be ideal in all situations. For example, the requirements for the application might change, such as shortening the response goal or allowing more time for priority instances. It is recommended that the timeout field is parameterized in the application code and populated via another means. This method prevents the need for the application to be recompiled if the timeout value is required to change.

Common mechanisms for this process are to parameterize each timeout value in the application code and to read in the values during execution time, from controlled record stores, such as properties and VSAM files, and from database records.

For ease of demonstration of the timeout feature, this scenario uses a CICS temporary storage (TS) queue to provide the WEBHOME program with the timeout value. The name of the loan timeout TS queue is LTIMEOUT. The timeout value is specified in milliseconds.

To provide the timeout value for the loan service, a new TS queue is populated with the required 5-second value.

8. Issue the following command on the CICS terminal screen:

```
CECI WRITEQ TS QUEUE('LTIMEOUT') FROM('5000')
```

Now, you have a TS queue named LTIMEOUT that contains the value 5000, which will be used by the WEBHOME program to populate the TIMEOUT parameter on a **FETCH CHILD** command.

### 5.2.5 Adding the TIMEOUT parameter to the FETCH command of the unreliable service

Next, update the WEBHOME program to parameterize a TIMEOUT parameter on the **FETCH CHILD** command of the GETLOAN service. Complete the following tasks:

- ▶ Add new record structures to reference the method of obtaining the timeout value. In this example, read the value from a LTIMEOUT TS queue.
- ▶ Add application logic to read the timeout value from the TS queue.
- ▶ Set the TIMEOUT parameter on the **FETCH CHILD** command.
- ▶ Update the behavior of the **FETCH CHILD** logic to react appropriately to a timeout response.
- ▶ Display status messages to log the behaviour of the WEBHOME program.

#### Add the record structures to hold the timeout value

9. Add the code shown in **bold** font in Example 5-7 to the working storage section of the WEBHOME program.

*Example 5-7 Records to manage the Loan quote service timeout*

---

|                                                                       |                          |                                        |
|-----------------------------------------------------------------------|--------------------------|----------------------------------------|
| 1                                                                     | COMMAND-RESP             | PIC S9(8) COMP.                        |
| 1                                                                     | COMMAND-RESP2            | PIC S9(8) COMP.                        |
| <br><b>* Record for TSQ containing timeout details for loan quote</b> |                          |                                        |
| 1                                                                     | <b>TIMEOUT-TSQ.</b>      |                                        |
| 2                                                                     | <b>TSQ-NAME</b>          | <b>PIC X(8) VALUE 'LTIMEOUT'.</b>      |
| 2                                                                     | <b>TSQ-TIMEOUT</b>       | <b>PIC X(8) VALUE ' '.</b>             |
| 2                                                                     | <b>TIMEOUT-LEN</b>       | <b>PIC S9(4) USAGE BINARY.</b>         |
| 1                                                                     | <b>LOAN-RATE-TIMEOUT</b> | <b>PIC S9(8) USAGE BINARY VALUE 0.</b> |

---

These new records are used to store and issue the timeout value for the loan quote service.

## Read timeout value from external source

After running a child task to initiate the GETLOAN service, next read the required timeout value from an external data source. For this example, use the TS queue LTIMEOUT.

10. Add the code shown in **bold** font in Example 5-8 to the WEBHOME program to obtain a timeout value from an external source.

*Example 5-8 Reading a value from a TS queue at execution time*

---

```
EXEC CICS RUN TRANSID (GETLOAN-TRAN)
 CHANNEL (MYCHANNEL)
 CHILD (GET-LOAN-TKN)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

* -----
* Before fetching (and blocking) on the loan quote results
* Check to see if we should apply a TIMEOUT.
* Typically from a FILE or DB2 look up -
* for simplicity we will use a TSQ.
* -----
MOVE 8 TO TIMEOUT-LEN
EXEC CICS READQ TS QUEUE (TSQ-NAME)
 ITEM (1)
 INTO (TSQ-TIMEOUT)
 LENGTH (TIMEOUT-LEN)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

IF COMMAND-RESP = DFHRESP(NORMAL)
THEN

* -----
* Found a timeout value to use on the FETCH of the quote
* -----
MOVE TSQ-TIMEOUT(1:TIMEOUT-LEN) TO LOAN-RATE-TIMEOUT

INITIALIZE STATUS-MSG
STRING 'Timeout of '
 DELIMITED BY SIZE
 TSQ-TIMEOUT
 DELIMITED BY SPACE
 ' milliseconds to get loan rate quote.'
 DELIMITED BY SIZE
 INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

ELSE

* -----
* Did not find a timeout value. Continue with NO timeout
* A TIMEOUT(0) parameter on the FETCH indicates no timeout
```



```

* -----

MOVE 0 TO LOAN-RATE-TIMEOUT

INITIALIZE STATUS-MSG
MOVE 'Timeout not set for loan rate quote.' TO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE
END-IF

* -----
* Perform the FETCH of loan rate
* -----

```

---

The code extract in Example 5-8 on page 66 sets the LOAN-RATE-TIMEOUT field to the number that is specified in TS queue LTIMEOUT. If there is a failure to read the TS queue, the value of '0' is used for the timeout.

**Note:** A TIMEOUT of '0' (zero) on the **FETCH** commands indicates that the TIMEOUT parameter is not set, rather than waiting for a length of 0 milliseconds. If the intention is to code a non-blocking **FETCH** command, use the **NOSUSPEND** parameter. See “Tip: Check the status of a child without blocking the parent by using the NOSUSPEND option” on page 118 for an example of using the **NOSUSPEND** option.

### Add the timeout parameter to the FETCH command

In the WEBHOME program, you obtained the required timeout value from an external source. Now use this value to parameterize the TIMEOUT parameter for the **FETCH** command of the loan quote child task. Also add handling code for the **FETCH CHILD** command to check for a timeout response.

11. As shown in Example 5-9 add the code shown in **bold** font to the WEBHOME program to add a timeout parameter to the retrieval of the personalized loan quote service.

**Note:** Do not forget the END-IF statement near the end of Example 5-9.

*Example 5-9 Add a timeout parameter and check response codes for the FETCH CHILD command*

---

```

* -----
* Perform the FETCH of loan rate
* -----

EXEC CICS FETCH CHILD (GET-LOAN-TKN)
 TIMEOUT (LOAN-RATE-TIMEOUT)
 CHANNEL (GET-LOAN-CHAN)
 COMPSTATUS (CHILD-RETURN-STATUS)
 ABCODE (CHILD-RETURN-ABCODE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

* -----
* Check if the FETCH of the child's results timed out
* -----

IF COMMAND-RESP = DFHRESP(NOTFINISHED) AND COMMAND-RESP2 = 53
THEN
 INITIALIZE STATUS-MSG

```

```

MOVE
 'Abandoned loan quote because it took too long!'
 TO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

ELSE

 PERFORM CHECK-COMMAND
 PERFORM CHECK-CHILD

* -----
* Successful response from the child.
* Get the personalized loan quote
* -----
EXEC CICS GET CONTAINER (GETLOAN-CONTAINER)
 CHANNEL (GET-LOAN-CHAN)
 INTO (CUSTOMER-LOAN-RATE)
 RESP (COMMAND-RESP)
 RESP2 (COMMAND-RESP2)

END-EXEC

PERFORM CHECK-COMMAND

* -----
* Finally, display the loan quote
* -----

INITIALIZE STATUS-MSG
STRING 'Personalized Loan Rate: '
 DELIMITED BY SIZE
 CUSTOMER-LOAN-RATE
 DELIMITED BY SPACE
 ' %'
 DELIMITED BY SIZE
 INTO MSG-TEXT
PERFORM PRINT-STATUS-MESSAGE

END-IF

* Send a message to the screen to
* notify terminal user of completion
MOVE 'COMPLETE' TO CURRENT-STATUS
PERFORM PRINT-TEXT-TO-SCREEN

```

---

Now, make the updated code available to CICS:

12. Compile the updated WEBHOME program to the data set library that is available to CICS.

13. Make the new version of the WEBHOME program available to your CICS region by issuing the following command in CICS:

```
CEMT SET PROGRAM(WEBHOME) NEW
```

You now have completed the source code updates to the WEBHOME program and have parameterize the TIMEOUT value on a **FETCH CHILD** command. You populated the timeout value with a value set from an external source, a TS queue. This process allows you to update the timeout value in the future without the need to edit and recompile the WEBHOME source code.

You have also added basic status messages to the WEBHOME program to report on the path that it is taking.

## 5.3 Running the updated application

After deploying the modified WEBHOME program with the updates in 5.2.5, “Adding the TIMEOUT parameter to the FETCH command of the unreliable service” on page 65, you can test the program by issuing the WEBH transaction, along with a customer number, such as 0001.

14. Issue the following transaction and customer number in the CICS terminal window:

```
WEBH 0001
```

The updated WEBHOME program executes and runs a child task to obtain a personalized loan quote, which is then fetched and displays as a status message in the CICS language environment messages log, such as CEEMSG. A timeout value is set on the retrieval of the quote to prevent exceeding the SLA.

Example 5-10 shows the results of this type of execution of the WEBHOME program. Important parts of the status messages are highlighted in **bold font**.

*Example 5-10 Messages from a timely response of the loan quote service*

---

```
TC96WEBH 20171009120418 12:04.18 Started Web banking log-on data retrieval
TC96WEBH 20171009120421 12:04.21 Welcome Pradeep Gohil
TC96WEBH 20171009120421 12:04.21 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
TC96WEBH 20171009120421 12:04.21 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
TC96WEBH 20171009120421 12:04.21 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
TC96WEBH 20171009120422 12:04.22 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
TC96WEBH 20171009120422 12:04.22 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
TC96WEBH 20171009120422 12:04.22 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
TC96WEBH 20171009120422 12:04.22 Timeout of 5000 milliseconds to get loan rate
 GETL 20171009120422 Loan quote service under normal load. ETA 4 secs.
TC96WEBH 20171009120426 12:04.26 Personalized Loan Rate: 1.25 %
TC96WEBH 20171009120426 12:04.26 Ended Web banking log-on data retrieval
```

---

The status messages written by the banking example show that the 5-second (5000 milliseconds) timeout is sufficient to receive a reply from the loan quote service. The response time of the web banking application is within the SLA for the application.

Example 5-11 shows the status messages from an invocation of the WEBHOME program to a slow-responding quote service.

*Example 5-11 Messages from a non-timely response of the loan quote service*

---

```
T124WEBH 20171009143535 14:35.35 Started Web banking log-on data retrieval
T124WEBH 20171009143538 14:35.38 Welcome Pradeep Gohil
T124WEBH 20171009143538 14:35.38 Acc: 20140720 Bal: £0.01 Overdraft: £0.00
T124WEBH 20171009143538 14:35.38 Acc: 25875343 Bal: £45742.00 Overdraft: £1000.00
T124WEBH 20171009143538 14:35.38 Acc: 20170125 Bal: £34533.23 Overdraft: £0.00
T124WEBH 20171009143539 14:35.39 Acc: 62837456 Bal: £234.56 Overdraft: £0.00
T124WEBH 20171009143539 14:35.39 Acc: 64620987 Bal: £3092.60 Overdraft: £1000.00
T124WEBH 20171009143539 14:35.39 Acc: 64563923 Bal: £10123.98 Overdraft: £0.00
T124WEBH 20171009143539 14:35.39 Timeout of 5000 milliseconds to get loan rate
 GETL 20171009143539 Loan quote service under heavy load. ETA 7 secs.
```

---

T124WEBH 20171009143544 14:35.44 **Abandoned loan quote because it took too long!**  
T124WEBH 20171009143544 14:35.44 Ended Web banking log-on data retrieval

Example 5-11 shows that, under heavy load, the response time of the loan service changes from 4 to 7 seconds, which previously had the undesirable effect of reducing the response time of the application to beyond the 9-second SLA. Now that a timeout is implemented on the time that the WEBHOME program is willing to wait for a response time from the GETLOAN service, the application maintains its SLA.

Figure 5-4 depicts the effect of the unreliable GETLOAN service against the response time goals of the WEBHOME program, with the addition of a timeout specified on the **FETCH CHILD** command.

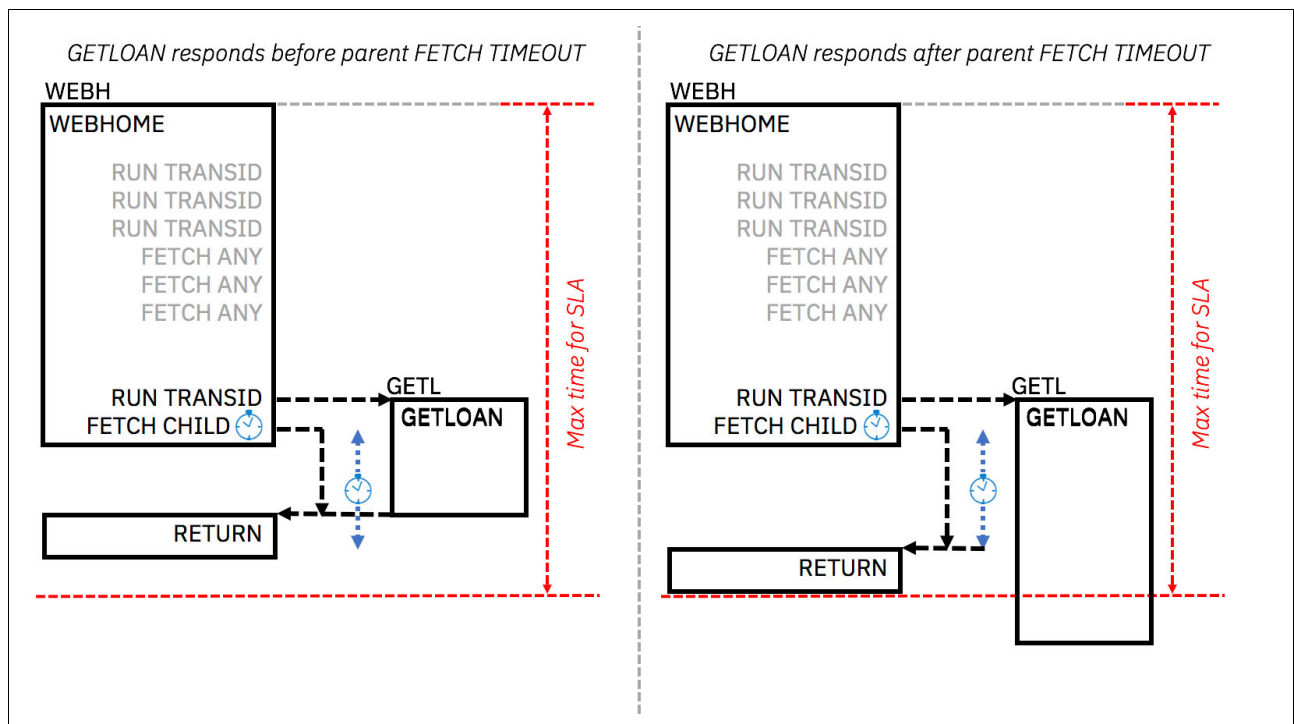


Figure 5-4 Specify a timeout value on the **FETCH CHILD** command to meet response time goals

Figure 5-4 shows the effect of adding the timeout in comparison to Figure 5-3 on page 64. The left diagram in Figure 5-4 shows that when the GETLOAN service replies in a timely manner, the WEBHOME program continues execution before the timeout value is met. The right diagram in Figure 5-4 shows that the timeout value is reached during the suspend of the **FETCH CHILD** command. This process enables the WEBHOME program to regain control and ultimately to complete (albeit without the GETLOAN service reply) within its response time SLA. Note also in the right diagram in Figure 5-4 that the GETLOAN service execution is unaffected by the timeout specified on the **FETCH CHILD** command, so there is no need to add extra logic to it to handle this case.

**Tip:** The GETLOAN service is coded to randomly return in 4 or 7 seconds. To force a timeout (or non-timeout) condition in the WEBHOME program or just to experiment with the **FETCH** command timeout behaviour, you can alter the timeout value to read in from the TS queue. Specify the timeout, in milliseconds, by issuing the following command in CICS (for example, setting a 9-second timeout in the WEBHOME program, and the GETLOAN child will not time out):

```
CECI WRITEQ TS QUEUE('LTIMEOUT') FROM('9000') ITEM(1) REWRITE
```

Alternatively, the following example sets a 1-second timeout in the WEBHOME program, and the GETLOAN child will always time out:

```
CECI WRITEQ TS QUEUE('LTIMEOUT') FROM('1000') ITEM(1) REWRITE
```

## 5.4 Summary

The response time savings of the prior chapters made it possible for the web banking example to be enhanced with new features. A new business opportunity, in the form of upselling the company's line of personal loans, required an update to the web banking example. However, maintaining customer satisfaction is still a key goal.

Due to the personalization requirements, the application update involved running a new child task that did not run concurrently with any other child tasks from the parent. Rather than relinquishing control by using a **LINK**, the example harnessed a **RUN TRANSID** command to maintain control and to act as a good-practice model for similar business opportunities in future.

The response time of the new service provider proved to be unreliable. On some occasions the provider responded in a timely manner, but in other instances it exceeded the SLA goal for the application.

The decision was made to consume the result if it responds in time or to abandon the venture if it takes too long. This method was achieved by using the **TIMEOUT** parameter on the **FETCH CHILD** command and **FETCH ANY** commands. The **TIMEOUT** parameter allows a parent to signify the length of time it is willing to wait for a response from a child task.

Using the CICS asynchronous API resulted in producing a robust application that met its response time SLA, despite calling an unreliable service.

If you've read this chapter and would like to experiment further, the completed code for the scenario can be found in [Chapter 5 End Tag](#).





## Creating a Java-based controller in a mixed-language environment

Thus far, this book has discussed the CICS asynchronous API in languages that make use of the EXEC CICS API, such as Common Business Oriented Language (COBOL). However, because CICS is a multi-language application server and one of those languages is Java, it is fitting that the scenarios in this book include a Java class library for CICS (JCICS) variant of the application programming interface (API). This variant allows a Java program in CICS to start another CICS task asynchronously and later collect the results of that child task. The child task can run programs written in any language CICS supports, such as COBOL, assembler, PL/I, C, and so on. Using channels and containers, these programs can receive data from, and pass data back to, the Java parents.

During development of the API, a standard Java approach was decided upon, making use of Java's Future interface, which is part of the `java.util.concurrent` package. Thus, any Java programmer who is familiar with the interface will understand the process. In addition, this approach allows any code written against this standard interface to work here. The CICS asynchronous API is particularly useful where existing business logic is used, which was written in languages supported in CICS other than Java. The API is supported within the IBM WebSphere® Liberty and Open Service Gateway Initiative (OSGi) runtimes when running in CICS.

This chapter explains Java-specific technology that is required for an understanding of the CICS asynchronous API, the mechanics of the Future interface, and generics. It then describes the new classes that are used to enable the full CICS asynchronous API in Java. Finally, a scenario converts the WEBHOME program written in COBOL in Chapter 5, “Developing robust applications with unreliable service providers” on page 55 to present a Liberty-based web front end, while utilizing existing business logic written in COBOL.

The code in the example continues in 6.3, “Providing a web front end for the web banking application” on page 80 and is provided on [GitHub](#).

## 6.1 Making promises about the future

The key to this Java implementation lies in the adoption of a standard Java interface, which comes with the Java virtual machine (JVM) itself, `java.util.concurrent.Future` (referred to as `Future` throughout this chapter). `Future` objects in Java are used to perform asynchronous activities. The CICS implementation allows those activities to be implemented in any programming language that is supported by CICS. The basic idea is that a task is performed—say, a method is invoked—to start an asynchronous workload. (This method by definition does not block the thread that invoked it.) The return type of this method is the `Future` interface, indicating to the programmer *in the future I promise to return a value*. You get to hold on to the `Future` and can exchange it later for the real value that you want. In Java, generics are employed here too, so that you know at compile time what the promised value's type will be, as illustrated in Figure 6-1.

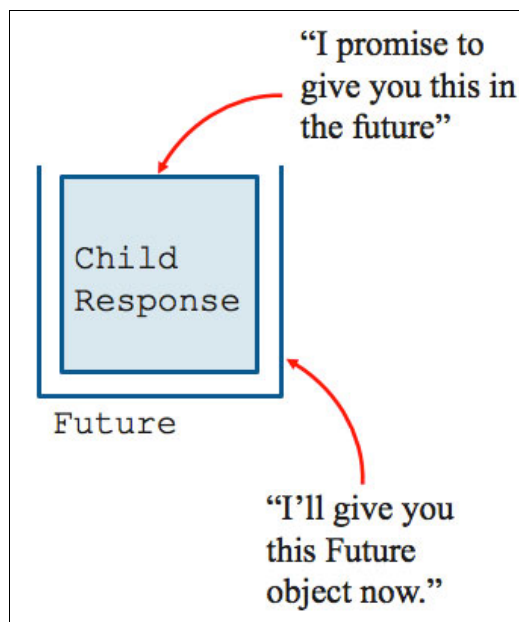


Figure 6-1 Java Futures and generics

Java's way of delivering this mechanism is to provide an interface, such that anyone wanting to make use of the interface has to provide an implementation that is fixed to that contract. The CICS asynchronous API provides an implementation of `Future` called `CICSFuture`. This implementation starts the asynchronous workload as a child task in the system, as usual, such that it's managed entirely by the CICS dispatcher.

**Note:** The concept of a child token, as returned by the **EXEC CICS RUN TRANSID** command, is encapsulated in the `CICSFuture`, so a Java programmer doesn't need to manage it.

To start a child task, you normally use the **RUN TRANSID** command. The Java equivalent of this command is as follows:

```
Future<ChildResponse> runTransactionId(String transactionId, Channel channel)
```



In this command, the second parameter, `channel`, is optional. If passed, this parameter is a standard JCICS `Channel` object in the `com.ibm.cics.server` package. You can store the return value of this method in a variable. Now that you have the promise of a result in the future, you can carry on with your work, knowing that you've started the child task. When you're interested in the results from the child task, you have the following choices:

- Check whether the child is back with results (mapping to the **FETCH CHILD NOSUSPEND** command):

```
boolean Future.isDone()
```

This method does not block but instead gives you a boolean value in return to tell you whether or not the child has completed. (Though, not only if the child has completed successfully. The child might have abended, and this method still returns `true`.) If the child task has yet to end, the method returns `false`.

- Block and wait for the child to return, if it hasn't already done so (mapping to the **FETCH CHILD** command):

```
ChildResponse Future.get()
```

This blocks until a result is returned from the child. When such a result is returned, it is in the form of a `ChildResponse` object. This result encapsulates the possible returned objects:

- The completion status of the child
- A `Channel` object, if the child returned a channel
- An abend code, if the child abended

Notice that this returned type matches with the generic type stated in the method signature of `runTransactionId()`.

The `get()` method also supports timeout, as the **FETCH CHILD** command does. To make use of this function, pass the method a long value and its time unit. For example, invoke `get(1, TimeUnit.SECONDS)` on a `Future` object to block until the result comes back, for a maximum of 1 second. The `TimeUnit` parameter is a built-in Java class in the `java.util.concurrent` package and has several enum values that are related to time duration, such as `MILLISECONDS`, `SECONDS`, `MINUTES`, and so on.

**Note:** The `CICSFuture` implementation that is provided also must support the `isCancelled()` and `cancel()` methods, given that the `Future` interface requires them. The method names suggest that the CICS task can be cancelled but the CICS asynchronous API does not allow the cancelling of a CICS task. Because neither of these operations are supported by the CICS asynchronous API, both methods throw `UnsupportedOperationException`.

Table 6-1 describes the meaning of the exceptions that can be thrown by `Future.get()`. `InterruptedException` is never thrown by the `CICSFuture` implementation, because there is no mechanism to interrupt a CICS task that's externalized beyond the CICS run time. However, it is required to be a possible `Exception` due to the `Future` contract.

*Table 6-1 Objects that can be returned or thrown by `Future.get()`*

| Class                                       | Description                                                                                |
|---------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>ChildResponse</code>                  | The returned object encompassing all the information that can be returned by a child task. |
| <code>java.lang.InterruptedException</code> | Never thrown by <code>CICSFuture</code> .                                                  |

|                                                      |                                                                                                |
|------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>java.util.concurrent.ExecutionException</code> | The child task has already been freed or the timeout value is invalid.                         |
| <code>java.util.concurrent.TimeoutException</code>   | Only thrown by <code>get(long timeout, TimeUnit unit)</code> . The child has not yet finished. |

This Future interface is the core mechanism behind the CICS asynchronous API in Java. The next section looks at the classes and methods that make up the Java implementation of the CICS asynchronous API in full, before continuing sections convert the WEBHOME program from Chapter 5, “Developing robust applications with unreliable service providers” on page 55 to Java.

## 6.2 CICS asynchronous API classes and methods

The first part of this section steps through a specific “golden-path” example. It shows in detail how to start a child task and collect any results it returns. The remainder of the section covers other methods that can be used.

### 6.2.1 A golden-path scenario

This scenario starts a child task and then collects its results. It uses the `AsyncService` and `Future` interfaces. Table 6-2 shows the concrete classes and their interfaces that are used in this section and throughout this chapter. All classes except `Future` are in the `com.ibm.cics.server` package. The API defines interfaces for each concrete class for the following reasons:

- To make changes to the implementation while fixing the defined contract provided to Java developers.
- To make mocking of objects easier and to ease testing.

*Table 6-2 Classes that make up the Java implementation of the CICS asynchronous API*

| Concrete class                 | Interface                                |
|--------------------------------|------------------------------------------|
| <code>AsyncServiceImpl</code>  | <code>AsyncService</code>                |
| <code>ChildResponseImpl</code> | <code>ChildResponse</code>               |
| <code>CICSFuture</code>        | <code>java.util.concurrent.Future</code> |

To start a new child task from a Java parent program, create a new instance of `AsyncServiceImpl`, and invoke the `runTransactionId()` method. This method is overloaded, such that a `Channel` can be optionally passed as an argument, but for now start a named transaction by passing that transaction name as the method’s argument, as shown in Example 6-1.

*Example 6-1 Invoking the `runTransactionId()` method*

```

AsyncService async = new AsyncServiceImpl();
Future<ChildResponse> child = null;
try
{
 child = async.runTransactionId("ABCD");
}
catch (InvalidRequestException e)
{

```

```

 System.out.println("Invalid request");
 }
 catch (InvalidTransactionIdException e)
 {
 System.out.println("Invalid transaction");
 }
 catch (NotAuthorisedException e)
 {
 System.out.println("Not authorised");
 }
 catch (ResourceDisabledException e)
 {
 System.out.println("disabled transaction error");
 }
}

```

On the first line, a new instance of `AsyncServiceImpl` is created, called `async`. This instance is required to call its methods. It is a stateless object, so it can be used multiple times within your CICS programs. The fewer stateless objects the better though, if for nothing more than tidiness of code. Next, create a variable to hold the `Future` information from the child. Then, start the child task by invoking `runTransactionId()` and passing the method the transaction name as a `String`. The method can throw a number of exceptions, as detailed in Table 6-3.

*Table 6-3 Objects that can be returned or thrown by `AsyncService.runTransactionId()`*

| Class                                      | Description                                                                                                                                                 |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Future</code>                        | The return type: a <code>Future</code> object with a promise to deliver a <code>ChildResponse</code> object.                                                |
| <code>InvalidRequestException</code>       | The started transaction is not shutdown-enabled. The CICS region is in the process of shutting down, or this method was run during transaction termination. |
| <code>InvalidTransactionIdException</code> | The transaction identifier specified is not defined to CICS or is defined as remote.                                                                        |
| <code>NotAuthorisedException</code>        | The user who is associated with the issuing task is not authorized to run the child task.                                                                   |
| <code>ResourceDisabledException</code>     | The specified transaction is disabled.                                                                                                                      |

To keep things simple here, write messages to the system log. The `runTransactionId()` method returns as soon as the new CICS task starts, almost immediately in a healthy system, so this Java program can now run any further logic that can be run concurrently with the child task. This basic example immediately fetches the child's response. To do this, use the `Future.get()` method against the child variable. Given that `runTransactionId()` returns the `Future<ChildResponse>` type, the return type of `child.get()` is of type `ChildResponse`. Because in this basic example the variable is used only inside the try block, declare response within the block, as shown in Example 6-2.

*Example 6-2 Fetching the child's response*

```

try
{
 ChildResponse response = child.get();
 if (response.getCompletionStatus().equals(CompletionStatus.NORMAL))
 {
 System.out.println("Child completed normally");
 }
}

```

```

 }
}
catch (InterruptedException e)
{}
catch (ExecutionException e)
{
 System.out.println("Child execution problem");
}

```

---

The two possible Exceptions are caught, and as before, a message is printed to the log only if one is thrown.

Taking a deeper look into the `ChildResponse` interface, it holds the following methods:

- ▶ `CompletionStatus getCompletionStatus()`
- ▶ `String getAbendCode()`
- ▶ `Channel getChannel()`

`CompletionStatus` is itself an enum, with the following possible values:

- ▶ `NORMAL`
- ▶ `ABEND`
- ▶ `SECERROR`

When a child task returns to its parent, you are guaranteed to have one of those possible states returned, and so `getCompletionStatus()` always returns a non-null value. If the child returns in a normal manner, `getCompletionStatus()` returns `CompletionStatus.NORMAL`. If the child abends, the method returns `CompletionStatus.ABEND`. The third possible state is unusual, but possible. The user security context is flowed from the parent task to the child task. Security checks occur in two places: as a child task is requested to be run, and just as the task is starting. If the CICS region is configured to not cache responses from the security manager, there is a small timing window where the first security check could pass, but the second could fail, causing the child task to not start. In this case, it's not truly an abend because the user program didn't actually start, so `CompletionStatus.SECERROR` is returned.

If the child abended, `getAbendCode()` returns the abend code as a `String`. If there was no abend, `null` is returned. Finally, if the child returns a channel, `getChannel()` returns a `JCICS Channel` object. Note that due to the mechanisms within the asynchronous API, to receive a channel from a child, you must have passed one in the first place.

In the example, if the completion status is normal, a message is printed to the log to say as much.

## 6.2.2 Additional methods: `getAny()` and `freeChild()`

Now that you've seen in detail how to start a child task and fetch its results, this section provides details about the following methods:

- ▶ `AsyncService.getAny()`
- ▶ `AsyncService.freeChild()`

The first method is straightforward. It maps to the **FETCH ANY** command. It's overloaded such that it can be invoked in the following ways:

- ▶ `ChildResponse getAny()`
- ▶ `ChildResponse getAny(BlockAction blockingAction)`
- ▶ `ChildResponse getAny(long timeout, TimeUnit unit)`

Each way returns a `ChildResponse` object, as with `CICSFuture.get()`. The `equals()` method is implemented here too, so you can compare a `ChildResponse` object to a `Future` object (backed by a `CICSFuture` implementation) to determine which child it was that returned.

The first `getAny()` method in this list blocks indefinitely, until any child previously started by that task returns.

**Note:** Child tokens are persisted across link levels, so it's possible for a COBOL program to start a child task, link to a Java program, and then issue `getAny()` and have the child return despite it not being started by the Java program.

If you want to effectively use the **FETCH ANY NOSUSPEND** command from within a Java program, `getAny(BlockingAction)` is the method to use. The parameter, `BlockingAction`, is an enum with possible values `SUSPEND`, `NOSUSPEND`. Specifying `SUSPEND` recreates the behavior of `getAny()` but is more explicit. Specifying `NOSUSPEND` simply checks whether a child task has returned and, if so, returns a `ChildResponse` object. If no child has returned, an `Exception` is thrown. You can statically import `BlockingAction.NOSUSPEND` to be able to use it naturally as shown in the following command:

```
import static com.ibm.cics.server.AsyncService.BlockingAction.NOSUSPEND;
...
async.getAny(NOSUSPEND);
```

Finally, a third overloaded method is available: `getAny(long timeout, TimeUnit unit)`. This method mirrors `Future.get(long timeout, TimeUnit unit)` and allows the invoking Java thread to be blocked until a child returns or until the timeout is hit, whichever is first. Table 6-4 documents the `Exceptions` that can be returned by the `getAny()` methods.

Table 6-4 Objects that can be returned or thrown by `AsyncService.getAny()`

| Class                                | Description                                                                                                                                          |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ChildResponse</code>           | Object representing the results of the first returned child task.                                                                                    |
| <code>InvalidRequestException</code> | The parent has no children.                                                                                                                          |
| <code>NotFinishedException</code>    | Only thrown by <code>getAny(long timeout, TimeUnit unit)</code> and <code>getAny(BlockingAction blockingAction)</code> . No children have completed. |
| <code>NotFoundException</code>       | No unfetched children were found.                                                                                                                    |

Finally, the `AsyncService.freeChild()` method is the overloaded method, which maps to the **FREE CHILD** command and can be passed a representation of a child task in the form of a `ChildResponse` or a `Future` (backed by a `CICSFuture` implementation). This process allows the invocation to take place before a child task returns and frees the CICS memory object that represents the specified child. Furthermore, when freed, `getAny()` no longer returns that child, and `Future.get()` against that specific child fails.

Table 6-5 lists the objects that can be returned by this method. The method frees only the CICS memory object that represents the specified child and does not garbage collect any `ChildResponse` object in the Java heap.

Table 6-5 Objects that can be returned or thrown by `AsyncService.freeChild()`

| Class                                | Description                                                                                    |
|--------------------------------------|------------------------------------------------------------------------------------------------|
| <code>void</code>                    | No response is returned.                                                                       |
| <code>InvalidRequestException</code> | The <code>Future</code> or <code>ChildResponse</code> object does not represent a valid child. |

Thus far, this chapter has covered most of the Java implementation of the CICS asynchronous API. The next section turns to a real-world example and continues the scenario built throughout this book.

## 6.3 Providing a web front end for the web banking application

This section continues the example built over the last few chapters. In this scenario, the parent program, `WEBHOME.cbl`, works well, but the company decides it requires a web-based front end for the system. There's a high cost in rewriting all the logic, so the company wants to reuse any code possible. Thus, this scenario uses the existing parent program, `WEBHOME.cbl`, and writes it in Java. In this manner, by taking advantage of Liberty in CICS and its support of JavaServer Pages (JSP), it is possible to quickly reach a working solution, while continuing to use the child programs written in COBOL.

### 6.3.1 Project setup

To make use of the Java classes described, you need to use IBM CICS Explorer® v5.4 or later. Using Explorer, follow these steps to set up the project:

1. Select **File** → **New** to create a new Dynamic web project. Call this project `AccountServices`.

By default this project will be the context root that Liberty uses in the URL for the resulting web content. You want to be able to deploy this project to a CICS Liberty server, and to do that you create two CICS bundles—one for the Liberty server and one for this project. (It's useful to be able to deploy the server independently to the content it serves.) To read more about how to use and deploy Liberty application server in CICS, see *IBM CICS and Liberty: What You Need to Know*, SG24-8335. After you define the Liberty server, a `JVMSEVER` resource is named.

2. Create a new CICS Bundle project and add a Dynamic web project includes resource.  
Choose the dynamic web project that you created earlier, and name the `JVMSEVER` to deploy it to.
3. Deploy this CICS Bundle project holding the Dynamic web project with the front-end logic you're about to write to z/OS file system (zFS).

After deployment, you can install the bundle into the CICS region in the usual manner.

## 6.3.2 Program architecture

Complete the following steps:

1. Under the WebContent folder in your Dynamic web project, create a `index.jsp` file.

This file presents a log in box for the customer to enter the customer account number and, in a fully-developed application, holds the authentication logic (see Figure 6-2 on page 81). For this example, keep it simple, and forward the entered customer number to the program to fetch the account details and other information.

2. Enter the HTML in the `index.jsp` file as shown in Example 6-3.

*Example 6-3 The `index.jsp` file*

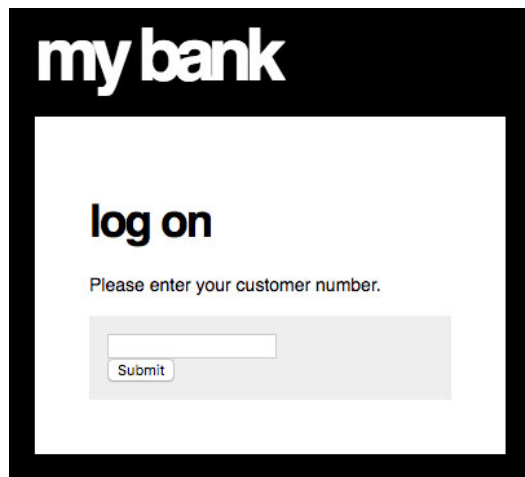
---

```
<h2>log on</h2>
<p>Please enter your customer number.</p>
<form action="CustomerAccounts.jsp" method="GET">
<input type="text" name="customer">

<input type="submit" value="Submit">
```

---

Clicking submit on this page makes a request to `CustomerAccounts.jsp?customer={number}` as shown in Figure 6-2.



*Figure 6-2 Log in page presented by Liberty*

Cascading Style Sheets (CSS) are added to the file to make the page look more presentable. (See the [GitHub](#) repository for more information).

3. Adjacent to `index.jsp`, create a new file called `CustomerAccounts.jsp`.

This file generates the HTML to present the customer account details and loan quote, which is the information that the WEBHOME program returns currently. You can write the HTML later, but for now write the logic to create the actual Java objects making the calls to the child programs.

4. Enter the code shown in Example 6-4 to `CustomerAccounts.jsp`.

*Example 6-4 `CustomerAccounts.jsp`*

---

```
<%@page import="banking.CustomerAccounts"%>
<%@page import="java.util.Map"%>
<%
 String customer = request.getParameter("customer");
 if (customer == null)
```

---

```

 {
 customer = "9999";
 }
 CustomerAccounts accountsPage = new CustomerAccounts(customer);
 Map<String, String> content = accountsPage.getContent();
%>

```

---

This code alludes to the fact that you are about to create a Java class, `CustomerAccounts`, in a package called `banking`, which is the parent program that calls the child programs. `CustomerAccounts.jsp` is just the visual front to this program. The code shown in Example 6-4 gets the `customer` parameter from the URL, checks whether it's actually present (and if not sets a default), and then creates a new instance of the `CustomerAccounts` class to pass the customer number to its constructor. The next section explains how this class works.

### 6.3.3 Writing the main program logic

The heart of the solution is the `CustomerAccounts` class in a package called `banking` in the `src` directory. Create the Java file with the constructor shown in Example 6-5.

*Example 6-5 CustomerAccounts class and constructor*

---

```

public class CustomerAccounts
{
 private byte[] accountNumber;
 public static String CODEPAGE = "CP037";
 public CustomerAccounts(String accountNumber)
 {
 try
 {
 // Ensure that the account number is 4 digits in length
 this.accountNumber =
 String.format("%04d", Integer.parseInt(accountNumber)).getBytes(
 CODEPAGE);
 }
 catch (UnsupportedEncodingException e)
 {
 e.printStackTrace();
 }
 }
}

```

---

The constructor takes the account number as a `String` and converts it to a byte array that represent the string in an EBCDIC code page. The number is converted because the account number is passed to the program `GETNAME.cb1`, which expects the account number to be in that binary format. Because `GETNAME.cb1` also expects the account number to be 4 digits in length, left-padded with zeros, use `String.format()` to ensure this. (This process requires you to quickly convert the account number `String` to an `Integer`, before getting the string's bytes.)

#### Setup: Getting content from the child tasks

Now that you have a new instance of `CustomerAccounts`, you need to write the logic to invoke the child programs. The first thing that `WEBHOME.cb1` does is put the account number in a container on a channel that is passed to all the child programs. Create a new method to this



class, `getContent()`, that can be invoked to return a Map that consists of the different details that the child programs return. The contents of this map can then be used when rendering the accounts page.

Add the method shown in Example 6-6 to `CustomerAccounts.java`.

*Example 6-6 The `getContent()` method*

---

```
public Map<String, String> getContent()
{
 final String accountPartnerTran = "PTNR";
 final String getCustomerNameTran = "GETN";
 final String getCurrentAccountTran = "ACUR";
 final String getLoanTran = "GETL";
 final String inputContainer = "INPUTCONTAINER";
 Channel myChannel = null;
 Set<Future<ChildResponse>> childTasks = HashSet<Future<ChildResponse>>();
 AsyncService async = new AsyncServiceImpl();
 Map<String, String> returnContent = new HashMap<String, String>();
 Task t = Task.getTask();
 try
 {
 myChannel = t.createChannel("MYCHANNEL");
 myChannel.createContainer(inputContainer).put(accountNumber);
 }
 catch (ChannelErrorException | ContainerErrorException
 | InvalidRequestException | CCSIDErrorException
 | CodePageErrorException e)
 {
 e.printStackTrace();
 }
}
```

---

This method starts by defining some String variables to hold the transaction and container names that make the later code more readable. It also creates a Channel type variable and a Set to hold child tasks (to be fully explained later) and creates a new instance of AsyncServiceImpl in variable `async`. This variable holds the methods, such as `runTransactionId()`, to start child tasks. Finally, `getContent()` returns a Map<String, String> object, so that you can create a variable `returnContent` of the same type, to which you can add the returned details from each child program.

Note how this method creates a channel (by creating it against the current task) before creating the input container with the byte array account number inside it. These methods can return a number of channel and container-related exceptions. More generally it's a good idea to deal with these exceptions properly, but here, to reduce printed code, just have a stack trace printed if one is thrown.

## Starting the child tasks asynchronously

Next, three child tasks are started from `WEBHOME.cb1` to get the following information:

- ▶ Partner account details
- ▶ The customer's name
- ▶ Current account details

To start the child tasks:

1. Add the code shown in Example 6-7 for the partner account details.

*Example 6-7 Partner account details*

---

```
Future<ChildResponse> accountPartner = null;
try
{
 accountPartner = async.runTransactionId(accountPartnerTran, myChannel);
}
catch (InvalidRequestException | InvalidTransactionIdException
 | ChannelErrorException | NotAuthorisedException
 | ResourceDisabledException e)
{
 e.printStackTrace();
}
childTasks.add(accountPartner);
```

---

To explain Example 6-7, we first create a variable, `accountPartner`, to store the `Future` object that you are about to have returned. Within the `try` block, invoke `runTransactionId()` and pass the partner account transaction PTNR and the `Channel` that you just created. This channel, which contains the input container, is copied to the child program. Additionally, because you passed a channel, you can also receive a channel on a future `get()` or `getAny()` invocation. Also add this `Future` object to the `childTasks` set. Then, you can perform operations easily over all child tasks (as explained in the next section).

2. Repeat this code twice for the next two child tasks, with the `customerName` and `currentAccount` variables.

## Collecting results from the three child tasks

The next thing that `WEBHOME.cbl` does is issue the **FETCH ANY** command within a loop, looping three times, to get the results from all three child tasks. Because the order in which these child tasks return is indeterminate (because there is no guarantee how long each one will take to run), you have to check which child's results was returned each time, and then run particular logic for each child.

Enter the code shown in Example 6-8 to perform this logic.

*Example 6-8 Collect results from the three child tasks*

---

```
ChildResponse anyResponse = null;
Iterator<Future<ChildResponse>> childIterator = childTasks.iterator();
while (childIterator.hasNext())
{
 try
 {
 anyResponse = async.getAny();

 if (anyResponse.equals(customerName))
 {
 returnContent.put("name",
 decodeContainerBytes(anyResponse, "GETNAMECONTAINER"));
 }
 else if (anyResponse.equals(currentAccount))
 {
 BankAccounts currentAccounts =
```

---

```

 new BankAccounts(anyResponse.getChannel()
 .getContainer("ACCTCURRCONT").getNoConvert());

 returnContent.put("current-accounts", currentAccounts.htmlRows());
 }
 else if (anyResponse.equals(accountPartner))
 {
 BankAccounts partnerAccounts =
 new BankAccounts(anyResponse.getChannel()
 .getContainer("ACCTPTNRCONT").getNoConvert());

 returnContent.put("partner-accounts", partnerAccounts.htmlRows());

 }
 else
 {
 System.out.println("*** Unknown child fetched ***");
 }
}
catch (InvalidRequestException | NotFoundException
 | ContainerErrorException | ChannelErrorException
 | CCSIDErrorException | CodePageErrorException e)
{
 e.printStackTrace();
}

childIterator.next();
}

```

---

The code shown in Example 6-8 on page 84 first creates a variable to hold the `ChildResponse` object that is returned from each child task. Then, you can understand why you added the child task `Future` objects to a set. You can create an `Iterator` object from it and use it to loop through all the child tasks. Then, unlike where the number of children (three) is hard-coded in the COBOL program, you have no need to and can observe the general preferred practice of not hard-coding values.

Inside a `try` block (to catch potential exceptions thrown), the logic invokes `async.getAny()` to fetch the first child task's results. You can then check this response with a set of `if` statements to perform logic based on which child returned. `ChildResponse` has the `equals()` method implemented and can be used to compare a `ChildResponse` to a `Future`. For example:

```
anyResponse.equals(customerName))
```

For each child that is returned, perform particular logic. If the returned child isn't recognized, write a message to the log. Finally, to reduce printed code, catch all the exceptions in bulk, and print a stack trace if any is thrown.

### ***Returning the customer's name***

If the returned child program is `GETNAME`, perform the following logic:

```
returnContent.put("name",
decodeContainerBytes(anyResponse, "GETNAMECONTAINER"));
```

In these commands, the code is simplified by moving some repeated code into the `decodeContainerBytes()` method.

Define the method shown in Example 6-9 in your class.

*Example 6-9 decodeContainerBytes() method*

---

```
private String decodeContainerBytes(ChildResponse response, String containerName)
{
 byte[] returnValue = null;

 try
 {
 returnValue = response.getChannel().getContainer(containerName).get();
 }
 catch (ContainerErrorException | ChannelErrorException
 | CCSIDErrorException | CodePageErrorException e)
 {
 System.out.println("Container error: " + containerName);
 }

 try
 {
 return new String(returnValue, CODEPAGE).trim();
 }
 catch (UnsupportedEncodingException e)
 {
 e.printStackTrace();
 return null;
 }
}
```

---

GETNAME.cb1 returns the name of the account holder as an EBCDIC-encoded string of bytes. decodeContainerBytes() takes the ChildResponse object, finds the container called GETNAMECONTAINER (in this case), which holds the account holder's name, and converts its contents to a Java String. The Container object's get() method returns a byte array, so you can create a new String from it (telling it the code page the bytes are in) and also use String.trim() to remove any whitespace on either side of the account holder's name.

Finally, within this if block we are inspecting in Example 6-8, add the account holder's name to the returnContent Map, against the key name.

### ***Returning the current and partner account details***

The other two child tasks return account information with the same data structure. Both ACCTCURR.cb1 and ACCTPTNR.cb1 return a binary structure that contains the number of accounts returned and, for each account, the account number, balance, and overdraft. In particular, the structure in ACCTCURR.cb1 that is returned in a container is shown in Example 6-10.

*Example 6-10 Structure in ACCTCURR.cb1*

---

2	NUMBER-OF-ACCOUNTS	PIC S9(4)	COMP-5	SYNC	VALUE 9.
2	ACCOUNT-DETAILS	OCCURS 5 TIMES.			
3	ACCT-NUMBER	PIC X(8)	VALUE	' '	.
3	BALANCE	PIC X(8)	VALUE	' '	.
3	OVERDRAFT	PIC X(8)	VALUE	' '	.

---

Thus, the first two bytes of the structure is the number of accounts. You want to take this information and convert it to a Java short (which takes up two bytes). The remainder—120

bytes—is the account details for all the accounts for the given customer. These details are in EBCDIC format. So, like earlier, convert this information to a Java String, noting the correct code page.

**Note:** For brevity in these examples and due to the fairly simple structure, we’ve written our own class to parse this structure. More generally, you can use [IBM Record Generator for Java](#) to convert generated ADATA files from COBOL copybooks or Assembler DSECTs to Java classes.

Because all this logic applies for the partner accounts to be returned also, you can create a BankAccounts class and have two instances of this class—one for the current accounts and the other for the partner accounts. Follow these steps:

1. Create a new Java class called BankAccounts.java that is adjacent to CustomerAccounts.java with the fields shown in Example 6-11.

*Example 6-11 BankAccounts class*

---

```
public class BankAccounts
{
 public short numberOfAccounts;
 private List<List<String>> accounts = new ArrayList<List<String>>();
}
```

---

The variable accounts stores the accounts that are returned by each child program as an ArrayList. Each account in turn is a List of strings, which includes the account number, balance, and overdraft.

2. Create a constructor that takes the raw bytes from a child’s returned container and populates these two fields, as shown in Example 6-12.

*Example 6-12 BankAccounts constructor*

---

```
public BankAccounts(byte[] input)
{
 ByteBuffer bb = ByteBuffer.allocate(2);
 bb.put(input, 0, 2);
 numberOfAccounts = bb.getShort(0);

 String details = null;
 try
 {
 details =
 new String(input, 2, 3 * 8 * numberOfAccounts,
 CustomerAccounts.CODEPAGE);
 }
 catch (UnsupportedEncodingException e)
 {
 e.printStackTrace();
 }

 for (short i = 0; i < numberOfAccounts; i++)
 {
 List<String> account = new ArrayList<String>();
 for (int j = 0; j < 3; j++)
 {
 account.add(details.substring((i * 24) + (j * 8),
```

```

 (i * 24) + ((j + 1) * 8)).trim());
 }
 accounts.add(account);
}
}

```

---

This constructor performs the following logic:

- Reads the number of accounts returned, creates a ByteBuffer object, and uses it to create a short.
- Converts the remaining bytes in the input to a String, using the appropriate code page.
- Populates the accounts list by looping over the account details for each account.
- Trims the strings also, because some of them aren't a full eight bytes of non-whitespace characters.

Back in `CustomerAccounts.getContent()`'s loop of reading the responses back from the children, use the `BankAccounts` class to parse the returned data.

3. For example, for current accounts, enter the logic shown in Example 6-13.

---

*Example 6-13 Current accounts logic*

---

```

else if (anyResponse.equals(currentAccount))
{
 BankAccounts currentAccounts =
 new BankAccounts(anyResponse.getChannel()
 .getContainer("ACCTCURRCONT").getNoConvert());
 returnContent.put("current-accounts",currentAccounts.htmlRows());
}

```

---

4. Repeat this logic for partner accounts too.

Within this logic, after you have an instance of `BankAccounts`, the `accounts` field is populated. The logic in Example 6-14 calls a method of this class, `htmlRows()`, which takes the accounts and produces an HTML string to format them as table rows, with the typical `<tr><td>` structure. For brevity, this method isn't printed here but is included in the `BankAccounts.java` on [GitHub](#).

5. To display this information, update `CustomerAccount.jsp` to print the information you've gathered so far, as shown in Example 6-14.

---

*Example 6-14 Display customer account information*

---

```

<p>Hello, <%=content.get("name")%>.</p>
<h2>my accounts</h2>
<h3>current accounts</h3>
<table>
<tr>
<td>Account number</td>
<td>Balance</td>
<td>Overdraft</td>
</tr>
<%=content.get("current-accounts")%>
</table>
<h3>other accounts</h3>
<table>
<tr>
<td>Account number</td>

```

```

 <td>Balance</td>
 <td>Overdraft</td>
 </tr>
 <%=content.get("partner-accounts")%>
</table>

```

The output of this information when run in Liberty is shown in Figure 6-3, styled with some CSS.

Hello, **Pradeep Gohil**.

## my accounts

Balances at 12:04:33 on 2017-11-07.

### current accounts

Account number	Balance	Overdraft
20140720	\$0.01	\$0.00
25875343	\$45742.00	\$1000.00
20170125	\$34533.23	\$0.00

### other accounts

Account number	Balance	Overdraft
62837456	\$234.56	\$0.00
64620987	\$3092.60	\$1000.00
64563923	\$10123.98	\$0.00

Figure 6-3 Name, current account, and partner account output on the accounts page

## Fetching a loan quote if time allows

The logic so far comprises all the information that must be returned for the customer to find use in the accounts page of the website. However, as with `WEBHOME.cb1`, you'll try and fetch a personalized loan quote if you have time. To do that, use `runTransactionId()` again to call the `GETLOAN` program, and then immediately wait for its results. However, you don't want to wait forever, so use a timeout parameter. `WEBHOME.cb1` reads this timeout value from a Temporary Storage (TS) queue, but a more Java-like process is to read this value from a properties file.

Follow these steps:

1. Add the following code to the `getContent()` method, as shown in Example 6-15.

Example 6-15 Additional code to `getContent()` method

```

long timeout = 0;
Properties prop = new Properties();
prop.setProperty("timeout", "0");
try
{
 FileInputStream input = new FileInputStream("async.properties");
 prop.load(input);
 if (prop.getProperty("timeout") != null)
 {
 timeout = Long.parseLong(prop.getProperty("timeout"));
 }
}

```

```

 }
 catch (IOException e)
 {
 System.out.println("Failed to find or read file async.properties");
 }
}

```

---

The timeout property is set to 0 as a default, before loading the properties file and reading that property from the file. (This scenario sets a default just in case the file cannot be read.) In this case, the file is read from WORK\_DIR, which for Liberty deployed into CICS using a CICS Bundle is defined in the jvmprofile file within the CICS Bundle.

2. Create the properties file, `async.properties`, in your Liberty WORK\_DIR. Ensure its file attributes make it readable to the user Liberty is running under.
3. Populate the properties file with the following timeout value:

```

milliseconds
timeout=50

```

4. In `CustomerAccounts.java`, start the child task to get the loan rate, as shown in Example 6-16.

---

*Example 6-16 Start the child task to get the loan rate*

---

```

Future<ChildResponse> loanRate = null;
try
{
 loanRate = async.runTransactionId(getLoanTran, myChannel);
}
catch (InvalidRequestException | InvalidTransactionIdException
 | ChannelErrorException | NotAuthorisedException
 | ResourceDisabledException e)
{
 e.printStackTrace();
}

```

---

5. Fetch the results from this child, using the timeout parameter shown in Example 6-17.

---

*Example 6-17 Fetch the loan rate*

---

```

ChildResponse returnedLoanRate = null;
try
{
 returnedLoanRate = loanRate.get(timeout, TimeUnit.MILLISECONDS);
}
catch (InterruptedException | ExecutionException e)
{
 e.printStackTrace();
}
catch (TimeoutException e)
{
 returnContent.put("loan-rate",
 "You could be eligible for great loan rates.");
 return returnContent;
}

```

---

If the child task doesn't end soon enough, a `TimeoutException` is thrown. In which case, you didn't manage to fetch a personalized loan rate, but you can at least return some generic text so that the web page doesn't have an empty spot. If there was no `Exception`



thrown, however, you know that the child task did complete in time. In which case, you need to confirm that the child task ended normally and did not abend.

6. Read the loan rate returned in a container, as shown in Example 6-18.

*Example 6-18 Read the loan rate*

---

```
if (returnedLoanRate.getCompletionStatus().equals(CompletionStatus.NORMAL))
{
 returnContent.put("loan-rate",
 "Great news! We've managed to obtain a great interest rate of "
 + decodeContainerBytes(returnedLoanRate, "GETLOANCONTAINER")
 + "% for you.");
}
return returnContent;
```

---

You make use of the enum `CompletionStatus` to confirm the loan rate child task's normal return state. In this case, you can store the personalized text for the web page in `returnContent` for the given customer. Because the data returned in the container by `GETLOAN` is also an EBCDIC string, use the method defined earlier, `decodeContainerBytes()`. Finally, return the Map you've been building up with content, `returnContent`.

You now have everything you need to construct the web page that details the account information and loan rate in `CustomerAccounts.jsp`.

### 6.3.4 Displaying the account details and loan rate

Add the code to `CustomerAccount.jsp` to print the loan rate (customized or generic) as shown in Example 6-19.

*Example 6-19 Print the loan rate*

---

```
<h2>need a loan?</h2>
<p><%=content.get("loan-rate")%></p>
<p id="smallprint">
Loan fetched using a timeout value of
 <%=content.get("timeout")%>ms.
</p>
```

---

Because `async.properties` that holds the timeout property value is read when generating this page, the value read (and used on the `get()` invocation for the loan rate) is printed in small print too (just for fun in this example). The right side of Figure 6-4 on page 92 shows that a short timeout period results in the child task not returning in time, and the generic loan rate text being printed. If the child task returns before the `TimeoutException` is thrown, the personalized loan rate is printed.

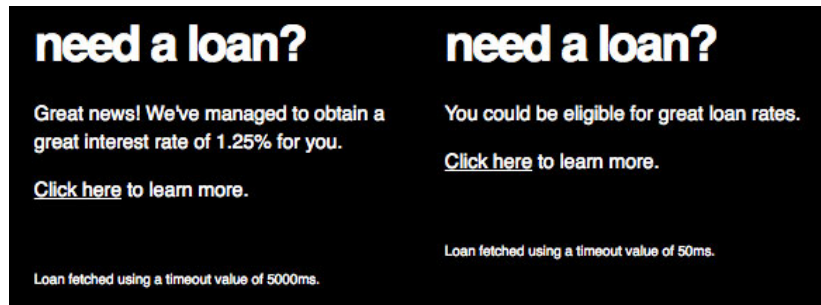


Figure 6-4 Different loan rate text being displayed, depending on if the child task completed in time.

Finally, Figure 6-5 shows the entire accounts page that is generated, with all the content fetched from the child tasks.

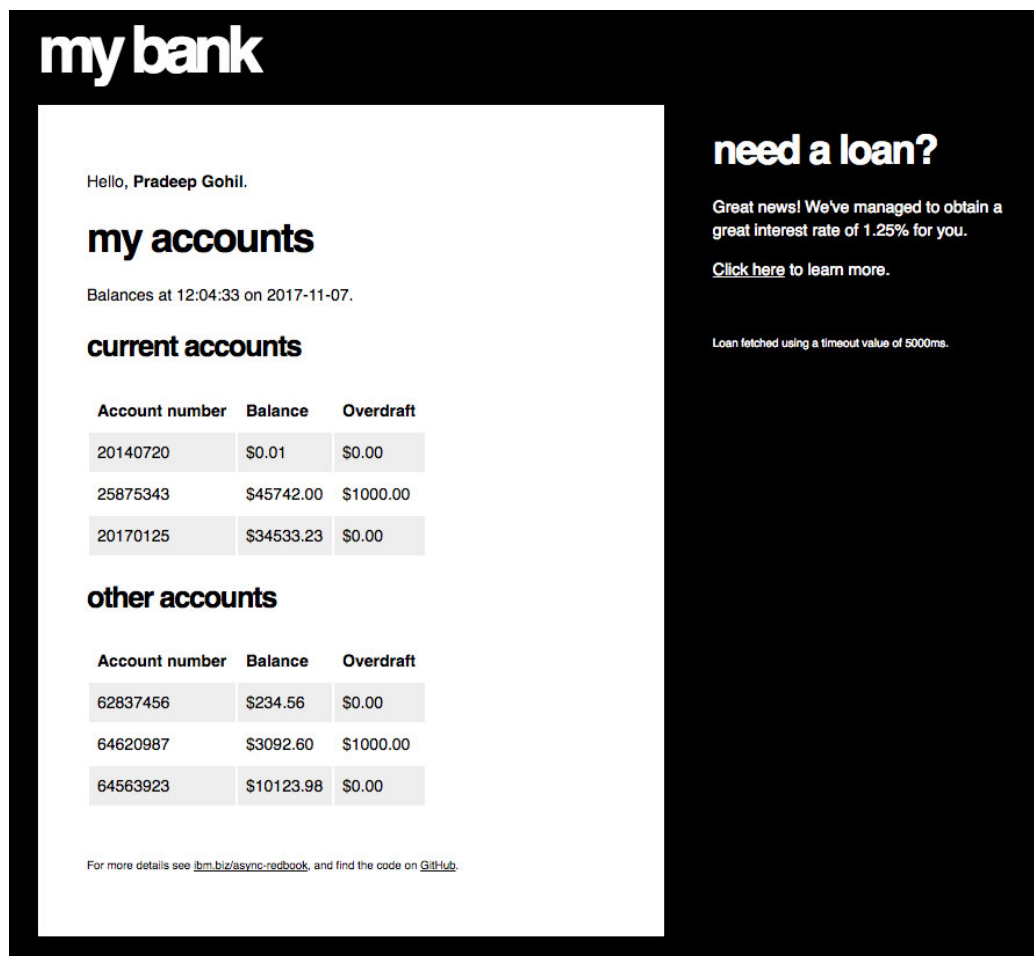


Figure 6-5 The complete generated accounts page.

## 6.4 Summary

This chapter described how to make use of the Java variant of the CICS asynchronous API. The API provides a full implementation of the API, while making it easy to consume for a Java programmer. As a set of JCICS classes, it runs in the OSGi and Liberty runtimes in CICS. When used with Liberty, you can now create CICS programs that serve web applications, while using existing business logic to minimize overall costs. This business logic can be run asynchronously to reduce load times.

You can find the full source code for the web banking application on [GitHub](#).





## Tips and tricks

This chapter provides techniques that you can use when using the asynchronous API with CICS applications. It includes the following tips and tricks:

- ▶ Trick: Reduce the management burden by running children under a single transaction ID
- ▶ Tip: Run existing COMMAREA-based assets asynchronously without changing them
- ▶ Tip: Release storage wisely in long-running parent transactions
- ▶ Trick: Prevent sets of children from interfering in FETCH ANY logic by using FREE CHILD
- ▶ Tip: Check the status of a child without blocking the parent by using the NOSUSPEND option
- ▶ Trick: Process as many children as possible in a fixed time period
- ▶ Tip: Using response-only channels between parent and child transactions

## 7.1 Trick: Reduce the management burden by running children under a single transaction ID

Previous examples have shown parent transaction where each of its children had its own transaction ID. An asynchronous API application has a single parent transaction and multiple children. Some of these children can perform similar activities (for example, obtaining summaries of different bank accounts held by an individual).

If these child applications are deployed conventionally, each child needs its own transaction ID, which then runs a designated child program. However, the child applications can all run under the same transaction ID, providing the transaction runs an initial *wrapper program* that dynamically picks which child program to link to. Such an application design requires an architected means for the parent application to convey the child program name to the child *wrapper program*.

This design follows this example:

- ▶ The parent transaction ID is PRNT, which runs the PARENT program.
- ▶ The child transaction ID is ASCH, which runs the ASYNCWP wrapper program.
- ▶ The parent program places a container, named CHILD-PROGRAM, on the channel that gets passed to a child transaction.
- ▶ This CHILD-PROGRAM container is extracted from the channel by the child wrapper program when the child transaction starts.
- ▶ The child wrapper program then links to the CHILD-PROGRAM by passing the channel that is provided by the parent.
- ▶ This example then runs two ASCH children that target the CHILD1 and CHILD2 programs.
- ▶ Then, the PARENT program runs. To see the code and an explanation of the PARENT program, go to 7.1.1, “The PARENT program running two different children under the ASCH child transaction ID” on page 97.
- ▶ Then, the ASYNCWP program runs. To see the code and explanation of ASYNCWP program, go to 7.1.2, “Using the ASYNCWP wrapper program to extract the target child program from a channel and linking to it” on page 102.
- ▶ The CHILD1 and CHILD2 programs run under the ASCH transaction. To see the code and explanation of this action, go to 7.1.3, “The CHILD1 and CHILD2 child programs running under the ASCH transaction” on page 103.

Figure 7-1 shows a pictorial representation of this scenario.

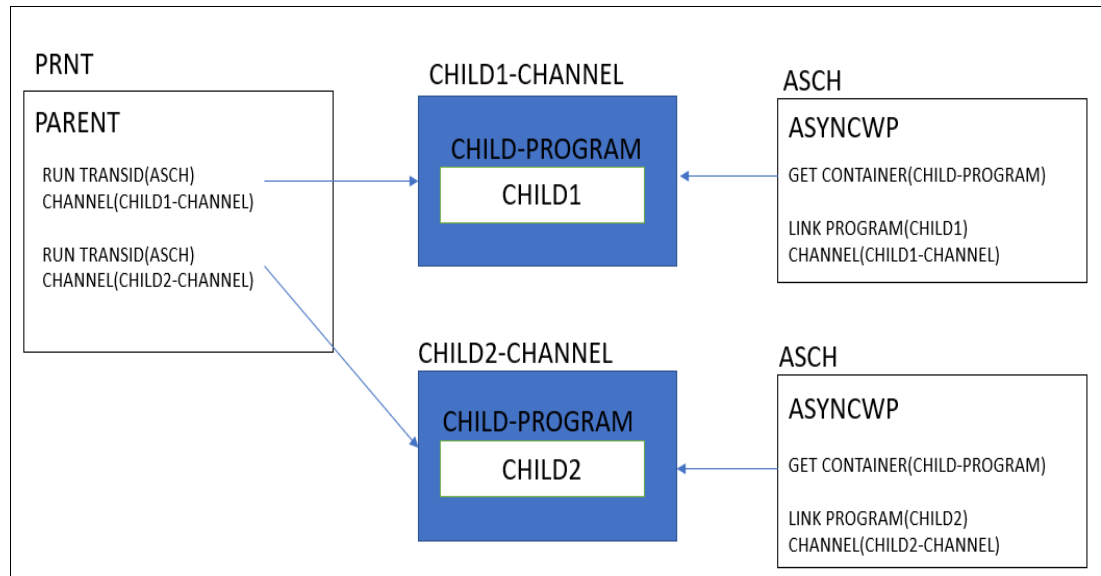


Figure 7-1 The PRNT transaction running two different ASCH children

### 7.1.1 The PARENT program running two different children under the ASCH child transaction ID

Example 7-1 shows the following information about the DATA DIVISION of the PARENT program:

- ▶ Declarations of the CHILD1 and CHILD2 child program names
- ▶ The text string that is passed to each child ('REQUEST FROM PARENT')
- ▶ The expected responses from the two child tasks
- ▶ Declarations to hold two child tokens used on the **RUN** and **FETCH** commands
- ▶ Declarations to hold two fetched channel names used on the **FETCH** commands and the **GET CONTAINER** commands

After a child is fetched, the response container is extracted using the **GET CONTAINER** command. In Example 7-1, the response is expected to be 30-bytes long.

Example 7-1 PARENT program DATA DIVISION

```

IDENTIFICATION DIVISION.
PROGRAM-ID. PARENT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 CONTAINER-LENGTH PIC S9(8) COMP.
01 CHILD-TOKENS.
03 CHILD1-TOKEN PIC X(16).
03 CHILD2-TOKEN PIC X(16).
01 CHILD-RESPONSE-CHANNELS.
03 CHILD1-RESPONSE-CHANNEL PIC X(16).
03 CHILD2-RESPONSE-CHANNEL PIC X(16).
01 CHILD-FETCH-STATE.
03 CHILD-COMPSTATUS PIC S9(8) COMP.

```

```

03 CHILD-ABCODE PIC X(4).
01 CHILD-PROGRAM-NAMES.
03 CHILD1-PROGRAM-NAME PIC X(8) VALUE 'CHILD1'.
03 CHILD2-PROGRAM-NAME PIC X(8) VALUE 'CHILD2'.
01 CHILD-EXPECTED-RESPONSES.
03 CHILD1-EXPECTED-RESPONSE PIC X(30)
 VALUE 'RESPONSE FROM CHILD1'.
03 CHILD2-EXPECTED-RESPONSE PIC X(30)
 VALUE 'RESPONSE FROM CHILD2'.
01 PARENT-REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 OPERATOR-MESSAGE PIC X(30)
 VALUE 'CHILD1 AND CHILD2 SUCCESSFUL'.
01 CHILD1-RESPONSE-POINTER POINTER.
01 CHILD1-RESPONSE-LENGTH PIC S9(8) COMP.
01 CHILD2-RESPONSE-POINTER POINTER.
01 CHILD2-RESPONSE-LENGTH PIC S9(8) COMP.
LINKAGE SECTION.
01 CHILD1-RESPONSE PIC X(30).
01 CHILD2-RESPONSE PIC X(30).

```

---

Example 7-2 shows the following information about the PARENT program's main routine section and its two RUN CHILD sections:

- ▶ The main routine calls the two RUN CHILD sections.
- ▶ The main routine then fetches the responses from the two children.
- ▶ The PARENT then terminates after sending a text message to the terminal operator.
- ▶ The channel created for each child contains a CHILD-PROGRAM container (used by the child wrapper program) and a child specific container used by the child business logic program (CHILD1-REQ and CHILD2-REQ).
- ▶ Each child runs as a ASCH transaction.

*Example 7-2 The PARENT program main routine section plus RUN CHILD sections*

---

```

PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
 PERFORM RUN-CHILD1-WITH-CHANNEL
 PERFORM RUN-CHILD2-WITH-CHANNEL
 PERFORM FETCH-CHILD1
 PERFORM FETCH-CHILD2
 EXEC CICS SEND TEXT FROM(OPERATOR-MESSAGE) ERASE

 FREEKB RESP(EXEC-RESP) END-EXEC
 EXEC CICS RETURN END-EXEC.
Main-Routine-End. EXIT.

***** **
RUN-CHILD1-WITH-CHANNEL *

 RUN-CHILD1-WITH-CHANNEL section.

```



```

*
EXEC CICS PUT CONTAINER('CHILD-PROGRAM')
 CHANNEL('CHILD1')
 FROM(CHILD1-PROGRAM-NAME)
 FLENGTH(LENGTH OF CHILD1-PROGRAM-NAME)
 RESP(EXEC-RESP) END-EXEC
EXEC CICS PUT CONTAINER('CHILD1-REQ')
 CHANNEL('CHILD1')
 FROM(PARENT-REQUEST-TEXT)
 FLENGTH(LENGTH OF PARENT-REQUEST-TEXT)
 RESP(EXEC-RESP) END-EXEC

EXEC CICS RUN TRANSID('ASCH') CHANNEL('CHILD1')
 CHILD(CHILD1-TOKEN) RESP(EXEC-RESP)
 END-EXEC.

RUN-CHILD1-WITH-COMMAREA-END. EXIT.

** RUN-CHILD2-WITH-CHANNEL *

 RUN-CHILD2-WITH-CHANNEL section.
*
 EXEC CICS PUT CONTAINER('CHILD-PROGRAM')
CHANNEL('CHILD2')
 FROM(CHILD2-PROGRAM-NAME)
 FLENGTH(LENGTH OF CHILD2-PROGRAM-NAME)
 RESP(EXEC-RESP) END-EXEC

EXEC CICS PUT CONTAINER('CHILD2-REQ')
 CHANNEL('CHILD2')
 FROM(PARENT-REQUEST-TEXT)
 FLENGTH(LENGTH OF PARENT-REQUEST-TEXT)
 RESP(EXEC-RESP) END-EXEC

EXEC CICS RUN TRANSID('ASCH') CHANNEL('CHILD2')
 CHILD(CHILD2-TOKEN) RESP(EXEC-RESP)
 END-EXEC.

RUN-CHILD2-WITH-COMMAREA-END. EXIT.

```

---

Example 7-3 on page 100 shows the following information about the **FETCH-CHILD1** section of the **PARENT** program:

- ▶ This example fetches the response channel and completion status of **CHILD1**. If the **FETCH** command fails or if **CHILD1** completes abnormally, the parent abends with abend code **CH1E**.
- ▶ After a successful **FETCH** command, the response container (**CHILD1-RESP**) is extracted from **CHILD1**'s response channel. If the **CHILD1-RESP** container does not exist or if it is the wrong length, the parent abends with abend code **CH1E**.
- ▶ Finally, the content of the extracted response container is examined. If the content is not correct, the parent abends with abend code **CH1E**.

*Example 7-3 PARENT FETCH-CHILD1 section*

---

```

** FETCH-CHILD1 *
** *
** FETCH CHILD1'S response channel then extract the *
** response container 'CHILD1-RESP'. Confirm that this *
** container holds the expected text *
** 'RESPONSE FROM CHILD1'. *

 FETCH-CHILD1 section.
 *
 EXEC CICS FETCH CHILD(CHILD1-TOKEN)
 CHANNEL(CHILD1-RESPONSE-CHANNEL)
 COMPSTATUS(CHILD-COMPSTATUS)
 ABCODE(CHILD-ABCODE)
 RESP(EXEC-RESP) END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-CH1E
 END-IF
 IF CHILD-COMPSTATUS NOT EQUAL DFHVALUE(NORMAL)
 PERFORM ABEND-CH1E
 END-IF

 EXEC CICS GET CONTAINER('CHILD1-RESP')
 CHANNEL(CHILD1-RESPONSE-CHANNEL)
 SET(CHILD1-RESPONSE-POINTER)
 FLENGTH(CHILD1-RESPONSE-LENGTH)
 RESP(EXEC-RESP)
 END-EXEC

 IF (EXEC-RESP NOT EQUAL DFHRESP(NORMAL)) OR
 (CHILD1-RESPONSE-LENGTH NOT EQUAL
 LENGTH OF CHILD1-EXPECTED-RESPONSE)
 PERFORM ABEND-CH1E
 END-IF

 SET ADDRESS OF CHILD1-RESPONSE TO
 CHILD1-RESPONSE-POINTER

 IF (CHILD1-RESPONSE NOT EQUAL
 CHILD1-EXPECTED-RESPONSE)
 PERFORM ABEND-CH1E
 END-IF.

 FETCH-CHILD1-END. EXIT.
```

---

Example 7-4 mirrors what appears in the FETCH-CHILD1 section shown in Example 7-3.

*Example 7-4 PARENT FETCH-CHILD2 section*

---

```

** FETCH-CHILD2 *
** *
** FETCH CHILD2'S response channel then extract the *
```

```

** response container 'CHILD2-RESP'. Confirm that this *
** container holds the expected text *
** 'RESPONSE FROM CHILD2'. *

 FETCH-CHILD2 section.
 *
 EXEC CICS FETCH CHILD(CHILD2-TOKEN)
 CHANNEL(CHILD2-RESPONSE-CHANNEL)
 COMPSTATUS(CHILD-COMPSTATUS)
 ABCODE(CHILD-ABCODE)
 RESP(EXEC-RESP) END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-CH2E
 END-IF
 IF CHILD-COMPSTATUS NOT EQUAL DFHVALUE(NORMAL)
 PERFORM ABEND-CH2E
 END-IF

 EXEC CICS GET CONTAINER('CHILD2-RESP')
 CHANNEL(CHILD2-RESPONSE-CHANNEL)
 SET(CHILD2-RESPONSE-POINTER)
 FLENGTH(CHILD2-RESPONSE-LENGTH)
 RESP(EXEC-RESP)
 END-EXEC

 IF (EXEC-RESP NOT EQUAL DFHRESP(NORMAL)) OR
 (CHILD2-RESPONSE-LENGTH NOT EQUAL
 LENGTH OF CHILD2-EXPECTED-RESPONSE)
 PERFORM ABEND-CH2E
 END-IF

 SET ADDRESS OF CHILD2-RESPONSE TO
 CHILD2-RESPONSE-POINTER

 IF (CHILD2-RESPONSE NOT EQUAL
 CHILD2-EXPECTED-RESPONSE)
 PERFORM ABEND-CH2E
 END-IF.

 FETCH-CHILD2-END. EXIT.

```

---

Example 7-5 shows the abend sections used by the PARENT program.

#### *Example 7-5 PARENT ABEND sections*

---

```

** ABEND-CH1E section - this section never returns *

 ABEND-CH1E section.
 EXEC CICS ABEND ABCODE('CH1E') END-EXEC.
 ABEND-CH1E-END. EXIT.

** ABEND-CH2E section - this section never returns *

```

```

ABEND-CH2E section.
EXEC CICS ABEND ABCODE('CH2E') END-EXEC.
ABEND-CH2E-END. EXIT.

```

---

### 7.1.2 Using the ASYNCWP wrapper program to extract the target child program from a channel and linking to it

Example 7-6 shows the following information about the ASYNCWP program, which is the asynchronous child wrapper program:

- ▶ This example extracts the CHILD-PROGRAM container from the channel passed by the parent transaction. If the container cannot be extracted, abend NOCH is issued.
- ▶ The LINK-CHILD-WITH-CHANNEL section obtains the name of the current channel (passed by the parent), which is saved in CHANNEL-NAME.
- ▶ The ASYNCWP program then links to the designated child program by passing the channel that the parent provided. If the link command fails due to an unknown program for example, the abend BADC is issued.
- ▶ When the designated child program returns, the ASYNCWP program terminates and the child task completes.
- ▶ There is no abend handling logic in the ASYNCWP program. If the designated child program abends with code XXXX, the child task terminates with that abend code. The XXXX abend code is then returned as the ABCODE when the parent fetches the child.

*Example 7-6 The ASYNCWP wrapper program*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. ASYNCWP.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 CHILD-PROGRAM PIC X(8).
01 CHANNEL-NAME PIC X(16).
LINKAGE SECTION.
PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
EXEC CICS GET CONTAINER('CHILD-PROGRAM')
 INTO(CHILD-PROGRAM)
 FLENGTH(LENGTH OF CHILD-PROGRAM)
 RESP(EXEC-RESP)
 END-EXEC
IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-NOCH
ELSE
 PERFORM LINK-CHILD-WITH-CHANNEL
END-IF
*
EXEC CICS RETURN END-EXEC.
Main-Routine-End. EXIT.

```

```

** LINK-CHILD-WITH-CHANNEL section *
** *
** LINK to CHILD-PROGRAM passing the current channel. *
** *
** We expect the child program to be CHANNEL AWARE. *
** It is the responsibility of the child program to place *
** any response container onto the channel which we pass *
** to it. *

LINK-CHILD-WITH-CHANNEL section.
EXEC CICS ASSIGN CHANNEL(CHANNEL-NAME)
RESP(EXEC-RESP)
END-EXEC
EXEC CICS LINK PROGRAM(CHILD-PROGRAM)
CHANNEL(CHANNEL-NAME)
RESP(EXEC-RESP) END-EXEC
IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
PERFORM ABEND-BADC
END-IF.
LINK-CHILD-WITH-CHANNEL-END. EXIT.

** ABEND-NOCH section - this section never returns *

ABEND-NOCH section.
EXEC CICS ABEND ABCODE('NOCH') END-EXEC.
ABEND-NOCH-END. EXIT.

** ABEND-BADC section - this section never returns *

ABEND-BADC section.
EXEC CICS ABEND ABCODE('BADC') END-EXEC.
ABEND-BADC-END. EXIT.

```

---

### 7.1.3 The CHILD1 and CHILD2 child programs running under the ASCH transaction

Example 7-7 shows the following information about the CHILD1 program:

- This example extracts the request container (name that is passed by the parent).
- If the request passed by the parent is correct, CHILD1 **PUTs** its response container onto the channel in container CHILD1-RESP, and then returns. If the request is not correct, the child issues the abend BADC.
- The **EXEC CICS RETURN** command returns to the ASYNCWP program, which also **RETURNS**.

*Example 7-7 Program CHILD1 with container response*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHILD1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

```

```

01 EXEC-RESP PIC S9(8) COMP.
01 REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 RESPONSE-TEXT PIC X(30)
 VALUE 'RESPONSE FROM CHILD1'.
01 REQUEST-AREA PIC X(30).
LINKAGE SECTION.
PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
 EXEC CICS GET CONTAINER('CHILD1-REQ')
 INTO(REQUEST-AREA)
 FLENGTH(LENGTH OF REQUEST-AREA)
 RESP(EXEC-RESP)
 END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-BADC
 END-IF

 IF REQUEST-AREA NOT EQUAL REQUEST-TEXT
 PERFORM ABEND-BADC
 END-IF

 EXEC CICS PUT CONTAINER('CHILD1-RESP')
 FROM(RESPONSE-TEXT)
 RESP(EXEC-RESP) END-EXEC

 EXEC CICS RETURN END-EXEC.

Main-Routine-End. EXIT.

** ABEND-BADC section - this section never returns *

ABEND-BADC section.
 EXEC CICS ABEND ABCODE('BADC') END-EXEC.
ABEND-BADC-END. EXIT.

```

---

Example 7-8 shows information about the CHILD2 program:

- ▶ This example extracts the request container (name that is passed by the parent).
- ▶ If the request passed by the parent is correct, CHILD2 **PUTs** its response container onto the channel in container CHILD2-RESP, and then returns. If the request is not correct the child issues an abend BADC.
- ▶ The **EXEC CICS RETURN** command returns to the ASYNCWP program, which also **RETURNS**.

*Example 7-8 Program CHILD2 with container response*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHILD2.
ENVIRONMENT DIVISION.

```

```

DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 RESPONSE-TEXT PIC X(30)
 VALUE 'RESPONSE FROM CHILD2'.
01 REQUEST-AREA PIC X(30).
LINKAGE SECTION.
PROCEDURE DIVISION.

** Main routine *

 Main-Routine section.
 *
 EXEC CICS GET CONTAINER('CHILD2-REQ')
 INTO(REQUEST-AREA)
 FLENGTH(LENGTH OF REQUEST-AREA)
 RESP(EXEC-RESP)
 END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-BADC
 END-IF

 IF REQUEST-AREA NOT EQUAL REQUEST-TEXT
 PERFORM ABEND-BADC
 END-IF

 EXEC CICS PUT CONTAINER('CHILD2-RESP')
 FROM(RESPONSE-TEXT)
 RESP(EXEC-RESP) END-EXEC

 EXEC CICS RETURN END-EXEC.

 Main-Routine-End. EXIT.

** ABEND-BADC section - this section never returns *

 ABEND-BADC section.
 EXEC CICS ABEND ABCODE('BADC') END-EXEC.
 ABEND-BADC-END. EXIT.

```

---

## 7.2 Tip: Run existing COMMAREA-based assets asynchronously without changing them

The asynchronous API uses a channel to pass state data between a parent and child program. Channels are more versatile than COMMAREAs (the other main method of passing state data between programs with CICS). There are circumstances where COMMAREAs have their uses.

For example, you might be re-deploying an existing COMMAREA-based synchronous application as an asynchronous API application. Such a re-deployment requires new logic to run at the parent level as new **RUN TRANSID** and **FETCH** commands will be needed. The parent needs to use a CHANNEL, because this is the only means of passing state data to a child. In addition, most of the logic that runs under the child need not be changed; however, it will need to handle a CHANNEL that is passed by the parent.

To avoid a significant re-coding of a COMMAREA-based child application to use channels and containers, you can use a *wrapper program*, which runs ahead of the child application. This wrapper program extracts a COMMAREA from the channel and links to the child program that is passing the extracted area as a COMMAREA.

This section shows this type of design, which is modelled on the wrapper program design shown in 7.1, “Trick: Reduce the management burden by running children under a single transaction ID” on page 96.

This design follows this example:

- ▶ The parent transaction ID is PRNT, which runs the PARENT program.
- ▶ The child transaction ID is ASCH, which runs ASYNCWP wrapper program.
- ▶ The parent places a container named CHILD-PROGRAM on the channel that gets passed to a child transaction.
- ▶ The parent places a container named REQUEST-COMM on the channel that gets passed to a child transaction, which is the COMMAREA.
- ▶ The CHILD-PROGRAM and REQUEST-COMM containers are extracted from the channel by the child wrapper program when the child transaction starts.
- ▶ The child wrapper program then links to the CHILD-PROGRAM, passing the REQUEST-COMM container contents as a COMMAREA.
- ▶ When CHILD-PROGRAM returns to the wrapper program, the response COMMAREA is placed on the channel as RESPONSE-COMM.
- ▶ This example runs two ASCH children, which target the CHILD1 and CHILD2 programs.
- ▶ Then, the PARENT program runs. To see the code and an explanation of the PARENT program, go to 7.2.1, “The PARENT program running two different children under child transaction ID ASCH passing COMMAREAs to each one” on page 107.
- ▶ Then, the ASYNCWP program runs. To see the code and explanation of ASYNCWP program, go to 7.2.2, “Using the ASYNCWP wrapper program to extract the PROGRAM target child from channel and linking to it with REQUEST-COMM COMMAREA” on page 112.
- ▶ The CHILD1 and CHILD2 programs run under the ASCH transaction. To see the code and explanation of this action, go to 7.2.3, “The CHILD1 and CHILD2 child programs running under the ASCH transaction” on page 114.



Figure 7-2 shows a pictorial representation of this scenario.

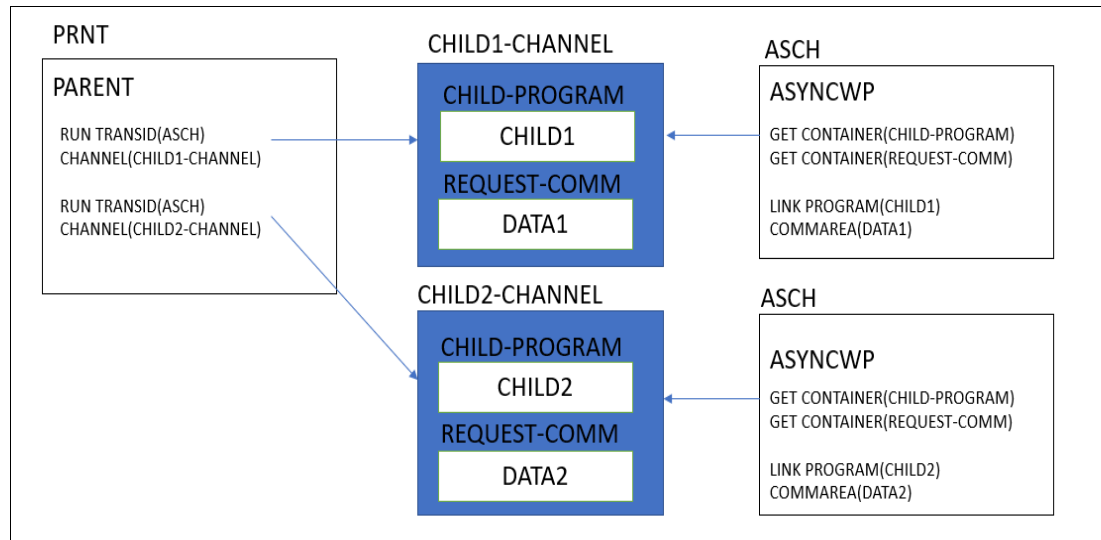


Figure 7-2 Transaction PRNT running two different ASCH children passing COMMAREAs

### 7.2.1 The PARENT program running two different children under child transaction ID ASCH passing COMMAREAs to each one

Example 7-9 shows the following information about the DATA DIVISION of the PARENT program:

- This example includes declarations of the child program names CHILD1 and CHILD2.
- It also contains the text string passed to each child ('REQUEST FROM PARENT')
- It includes the expected responses from the two child tasks.
- There are declarations to hold two child tokens used on the **RUN** and **FETCH** commands.
- There are declarations to hold two fetched channel names used on the **FETCH** commands and the **GET CONTAINER** commands.
- After a child is fetched, the response container is extracted using the **GET CONTAINER** command. In this example, the request and response for each child is a 60-bytes COMMAREA.

Example 7-9 PARENT program DATA DIVISION with COMMAREAs

```

IDENTIFICATION DIVISION.
PROGRAM-ID. PARENT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 CONTAINER-LENGTH PIC S9(8) COMP.
01 CHILD-TOKENS.
03 CHILD1-TOKEN PIC X(16).
03 CHILD2-TOKEN PIC X(16).
01 CHILD-RESPONSE-CHANNELS.
03 CHILD1-RESPONSE-CHANNEL PIC X(16).
03 CHILD2-RESPONSE-CHANNEL PIC X(16).
01 CHILD-FETCH-STATE.
03 CHILD-COMPSTATUS PIC S9(8) COMP.

```

```

03 CHILD-ABCODE PIC X(4).
01 CHILD1-COMMAREA.
03 CHILD1-REQUEST PIC X(30).
03 CHILD1-RESPONSE PIC X(30).
01 CHILD2-COMMAREA.
03 CHILD2-REQUEST PIC X(30).
03 CHILD2-RESPONSE PIC X(30).
01 CHILD-PROGRAM-NAMES.
03 CHILD1-PROGRAM-NAME PIC X(8) VALUE 'CHILD1'.
03 CHILD2-PROGRAM-NAME PIC X(8) VALUE 'CHILD2'.
01 CHILD-EXPECTED-RESPONSES.
03 CHILD1-EXPECTED-RESPONSE PIC X(30)
 VALUE 'RESPONSE FROM CHILD1'.
03 CHILD2-EXPECTED-RESPONSE PIC X(30)
 VALUE 'RESPONSE FROM CHILD2'.
01 PARENT-REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 OPERATOR-MESSAGE PIC X(30)
 VALUE 'CHILD1 AND CHILD2 SUCCESSFUL'.
LINKAGE SECTION.

```

---

Example 7-10 shows the following information about the PARENT program's main routine section and its two RUN CHILD sections:

- ▶ The main routine calls the two RUN CHILD sections.
- ▶ The main routine then fetches the responses from the two children.
- ▶ The PARENT program then terminates after sending a text message to the terminal operator.
- ▶ Each child channel is populated with a CHILD-PROGRAM container (used by the child wrapper program).
- ▶ Each child channel also contains a REQUEST-COMM container holding the child COMMAREA.
- ▶ Each child runs as an ASCH transaction.

*Example 7-10 PARENT program main routine section plus RUN CHILD COMMAREA sections*

---

```

PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
 PERFORM RUN-CHILD1-WITH-COMMAREA
 PERFORM RUN-CHILD2-WITH-COMMAREA
 PERFORM FETCH-CHILD1
 PERFORM FETCH-CHILD2
 EXEC CICS SEND TEXT FROM(OPERATOR-MESSAGE)
 ERASE FREEKB
 RESP(EXEC-RESP) END-EXEC
 EXEC CICS RETURN END-EXEC.
Main-Routine-End. Exit.

** RUN-CHILD1-WITH-COMMAREA *

```

\*\*\*\*\*

RUN-CHILD1-WITH-COMMAREA section.

\*

MOVE LOW-VALUES TO CHILD1-COMMAREA  
MOVE PARENT-REQUEST-TEXT TO CHILD1-REQUEST

EXEC CICS PUT CONTAINER('CHILD-PROGRAM')  
CHANNEL('CHILD1')  
FROM(CHILD1-PROGRAM-NAME)  
FLENGTH(LENGTH OF CHILD1-PROGRAM-NAME)  
RESP(EXEC-RESP) END-EXEC

EXEC CICS PUT CONTAINER('REQUEST-COMM')  
CHANNEL('CHILD1')  
FROM(CHILD1-COMMAREA)  
FLENGTH(LENGTH OF CHILD1-COMMAREA)  
RESP(EXEC-RESP) END-EXEC

EXEC CICS RUN TRANSID('ASCH') CHANNEL('CHILD1')  
CHILD(CHILD1-TOKEN) RESP(EXEC-RESP)  
END-EXEC.

RUN-CHILD1-WITH-COMMAREA-END. EXIT.

\*\*\*\*\*

\*\* RUN-CHILD2-WITH-COMMAREA \*

\*\*\*\*\*

RUN-CHILD2-WITH-COMMAREA section.

\*

MOVE LOW-VALUES TO CHILD2-COMMAREA  
MOVE PARENT-REQUEST-TEXT TO CHILD2-REQUEST

EXEC CICS PUT CONTAINER('CHILD-PROGRAM')  
CHANNEL('CHILD2')  
FROM(CHILD2-PROGRAM-NAME)  
FLENGTH(LENGTH OF CHILD2-PROGRAM-NAME)  
RESP(EXEC-RESP) END-EXEC

EXEC CICS PUT CONTAINER('REQUEST-COMM')  
CHANNEL('CHILD2')  
FROM(CHILD2-COMMAREA)  
FLENGTH(LENGTH OF CHILD2-COMMAREA)  
RESP(EXEC-RESP) END-EXEC

EXEC CICS RUN TRANSID('ASCH') CHANNEL('CHILD2')  
CHILD(CHILD2-TOKEN) RESP(EXEC-RESP)  
END-EXEC.

RUN-CHILD2-WITH-COMMAREA-END. EXIT.

---

Example 7-11 shows the following information about the FETCH-CHILD1 section of the PARENT program:

- This example fetches the response channel and completion status of CHILD1. If the **FETCH** command fails or if CHILD1 completed abnormally, the parent abends with abend code CH1E.
- After a successful **FETCH** command, the COMMAREA response container (RESPONSE-COMM) is extracted from CHILD1's response channel. If the RESPONSE-COMM container does not exist or if it is the wrong length, the parent abends with abend code CH1E.
- Finally, the content of the extracted response COMMAREA is examined. If the content is not correct, the parent abends with abend code CH1E.

*Example 7-11 PARENT FETCH-CHILD1 section fetches COMMAREA response*

---

```

** FETCH-CHILD1 *
** *
** FETCH CHILD1'S response channel then extract the *
** response commarea which is held in container *
** RESPONSE-COMM'. Confirm that the response commarea *
** holds the expected text 'RESPONSE FROM CHILD1'. *

 FETCH-CHILD1 section.
 *
 EXEC CICS FETCH CHILD(CHILD1-TOKEN)
 CHANNEL(CHILD1-RESPONSE-CHANNEL)
 COMPSTATUS(CHILD-COMPSTATUS)
 ABCODE(CHILD-ABCODE)
 RESP(EXEC-RESP) END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-CH1E
 END-IF
 IF CHILD-COMPSTATUS NOT EQUAL DFHVALUE(NORMAL)
 PERFORM ABEND-CH1E
 END-IF

 MOVE LENGTH OF CHILD1-COMMAREA TO CONTAINER-FLENGTH

 EXEC CICS GET CONTAINER('RESPONSE-COMM')
 CHANNEL(CHILD1-RESPONSE-CHANNEL)
 INTO(CHILD1-COMMAREA)
 FLENGTH(CONTAINER-FLENGTH) RESP(EXEC-RESP)
 END-EXEC

 IF (EXEC-RESP NOT EQUAL DFHRESP(NORMAL)) OR
 (CONTAINER-FLENGTH NOT EQUAL
 LENGTH OF CHILD1-COMMAREA)
 PERFORM ABEND-CH1E
 END-IF

 IF CHILD1-RESPONSE NOT EQUAL CHILD1-EXPECTED-RESPONSE
 PERFORM ABEND-CH1E
 END-IF.
 FETCH-CHILD1-END. EXIT.

```

---

Example 7-12 shows the FETCH-CHILD2 section of the PARENT program. This example mirrors what displays in the FETCH-CHILD1 section shown in Example 7-11 on page 110.

*Example 7-12 PARENT FETCH-CHILD2 section fetches COMMAREA response*

---

```

** FETCH-CHILD2 *
** *
** FETCH CHILD2'S response channel then extract the *
** response commarea which is held in container *
** RESPONSE-COMM'. Confirm that the response commarea *
** holds the expected text 'RESPONSE FROM CHILD2'. *

 FETCH-CHILD2 section.
*
 EXEC CICS FETCH CHILD(CHILD2-TOKEN)
 CHANNEL(CHILD2-RESPONSE-CHANNEL)
 COMPSTATUS(CHILD-COMPSTATUS)
 ABCODE(CHILD-ABCODE)
 RESP(EXEC-RESP) END-EXEC

 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-CH2E
 END-IF
 IF CHILD-COMPSTATUS NOT EQUAL DFHVALUE(NORMAL)
 PERFORM ABEND-CH2E
 END-IF

 MOVE LENGTH OF CHILD2-COMMAREA TO CONTAINER-FLENGTH

 EXEC CICS GET CONTAINER('RESPONSE-COMM')
 CHANNEL(CHILD2-RESPONSE-CHANNEL)
 INTO(CHILD2-COMMAREA)
 FLENGTH(CONTAINER-FLENGTH) RESP(EXEC-RESP)
 END-EXEC

 IF (EXEC-RESP NOT EQUAL DFHRESP(NORMAL)) OR
 (CONTAINER-FLENGTH NOT EQUAL
 LENGTH OF CHILD2-COMMAREA)
 PERFORM ABEND-CH2E
 END-IF

 IF CHILD2-RESPONSE NOT EQUAL CHILD2-EXPECTED-RESPONSE
 PERFORM ABEND-CH2E
 END-IF.

 FETCH-CHILD2-END. EXIT.

```

---

Example 7-13 shows the abend sections used by program PARENT.

*Example 7-13 PARENT ABEND sections*

---

```

** ABEND-CH1E section - this section never returns *

```

```

ABEND-CH1E section.
EXEC CICS ABEND ABCODE('CH1E') END-EXEC.
ABEND-CH1E-END. EXIT.

```

```

** ABEND-CH2E section - this section never returns *

ABEND-CH2E section.
EXEC CICS ABEND ABCODE('CH2E') END-EXEC.
ABEND-CH2E-END. EXIT.

```

---

## 7.2.2 Using the ASYNCWP wrapper program to extract the PROGRAM target child from channel and linking to it with REQUEST-COMM COMMAREA

Example 7-14 shows the following information about the ASYNCWP program, which is the asynchronous child wrapper program:

- ▶ This extracts the CHILD-PROGRAM container from the channel that is passed by the parent transaction. If the container cannot be extracted, abend NOCH is issued.
- ▶ It then extracts the REQUEST-COMM COMMAREA container from the channel that is passed by the parent transaction. If the container cannot be extracted, abend NOC0 is issued.
- ▶ If both containers are present, the LINK-CHILD-WITH-COMMAREA section is called.

*Example 7-14 ASYNCWP wrapper program extracting child program name and COMMAREA*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. ASYNCWP.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 COMMAREA-PTR POINTER.
01 COMMAREA-LEN.
03 FILLER PIC S9(4) COMP.
03 COMMAREA-LINK-LEN PIC S9(4) COMP.
01 CHILD-PROGRAM PIC X(8).
01 CHANNEL-NAME PIC X(16).
LINKAGE SECTION.
01 CHILD-COMMAREA PIC X(1).
PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
EXEC CICS GET CONTAINER('CHILD-PROGRAM')
 INTO(CHILD-PROGRAM)
 FLENGTH(LENGTH OF CHILD-PROGRAM)
 RESP(EXEC-RESP)
 END-EXEC
IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-NOCH
END-IF

```

```

** Get the request commarea out of the passed channel. *
** If there is no commarea issue an abend. *

 EXEC CICS GET CONTAINER('REQUEST-COMM')
 SET(COMMAREA-PTR)
 FLENGTH(COMMAREA-LEN) RESP(EXEC-RESP)
 END-EXEC
 IF EXEC-RESP EQUAL DFHRESP(NORMAL)
 PERFORM LINK-CHILD-WITH-COMMAREA
 ELSE
 PERFORM ABEND-NOCO
 END-IF
*
 EXEC CICS RETURN END-EXEC.
Main-Routine-End. EXIT.

```

---

Example 7-15 shows the following information about the ASYNCWP program linking to the child program passing the COMMAREA that is extracted from the channel passed by the parent. It also shows the abend routines at the end:

- ▶ If the link command fails, for example due to an unknown program, an abend BADC is issued.
- ▶ When the child program returns, the ASYNCWP program puts the response COMMAREA into a **RESPONSE-COMM** container and the child task completes.
- ▶ There is no abend handling logic in the ASYNCWP program. If the designated child program abends with code XXXX, the child task terminates with that abend code. The XXXX abend code then returned as the ABCODE when the parent fetches the child.

*Example 7-15 ASYNCWP links to child program with request COMMAREA then PUTs response COMMAREA onto channel*

---

```

** LINK-CHILD-WITH-COMMAREA section *
** *
** LINK to CHILD-PROGRAM passing the commarea addressed by *
** COMMAREA-PTR, length COMMAREA-LINK-LEN. *
** COMMAREA-LINK-LEN IS THE LOW ORDER 2-BYTES OF *
** COMMAREA-LEN. *
** This code could check that COMMAREA-LEN is within *
** acceptable limits but it currently assumes that the *
** parent task has passed a correct commarea. *

 LINK-CHILD-WITH-COMMAREA section.
 SET ADDRESS OF CHILD-COMMAREA TO COMMAREA-PTR
 EXEC CICS LINK PROGRAM(CHILD-PROGRAM)
 COMMAREA(CHILD-COMMAREA)
 LENGTH(COMMAREA-LINK-LEN)
 RESP(EXEC-RESP) END-EXEC
 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 PERFORM ABEND-BADC
 END-IF

** Now place the updated commarea onto our channel as *
** container RESPONSE-COMM. *

```

```

** This can be retrieved by the parent task when it fetches*
** the channel later on. *

EXEC CICS PUT CONTAINER('RESPONSE-COMM')
 FROM(CHILD-COMMAREA)
 FLENGTH(COMMAREA-LEN)
 RESP(EXEC-RESP) END-EXEC.
LINK-CHILD-WITH-COMMAREA-END. EXIT.

** ABEND-NOCH section - this section never returns *

ABEND-NOCH section.
EXEC CICS ABEND ABCODE('NOCH') END-EXEC.
ABEND-NOCH-END. EXIT.

** ABEND-NOCO section - this section never returns *

ABEND-NOCO section.
EXEC CICS ABEND ABCODE('NOCO') END-EXEC.
ABEND-NOCO-END. EXIT.

** ABEND-BADC section - this section never returns *

ABEND-BADC section.
EXEC CICS ABEND ABCODE('BADC') END-EXEC.
ABEND-BADC-END. EXIT.

```

---

### 7.2.3 The CHILD1 and CHIL2 child programs running under the ASCH transaction

Example 7-16 shows the following information about the CHILD1 program:

- ▶ If the request COMMAREA passed by the parent is correct, CHILD1 places its response into the COMMAREA then returns. If the request is not correct, the child issues an abend BADC.
- ▶ The **EXEC CICS RETURN** command returns to the ASYNCWP program.

*Example 7-16 Program CHILD1 with COMMAREA response*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHILD1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 RESPONSE-TEXT PIC X(30)
 VALUE 'RESPONSE FROM CHILD1'.

LINKAGE SECTION.
01 DFHCOMMAREA.
 03 REQUEST-AREA PIC X(30).
 03 RESPONSE-AREA PIC X(30).

PROCEDURE DIVISION.

```



```

** Main routine *

Main-Routine section.
*
 IF EIBCALEN NOT EQUAL LENGTH OF DFHCOMMAREA
 PERFORM ABEND-BADC
 END-IF

 IF REQUEST-AREA NOT EQUAL REQUEST-TEXT
 PERFORM ABEND-BADC
 END-IF

 MOVE RESPONSE-TEXT to RESPONSE-AREA

 EXEC CICS RETURN END-EXEC.

Main-Routine-End. EXIT.

** ABEND-BADC section - this section never returns *

ABEND-BADC section.
 EXEC CICS ABEND ABCODE('BADC') END-EXEC.
ABEND-BADC-END. EXIT.

```

---

Example 7-17 shows the following information about the CHILD2 program:

- If the request COMMAREA passed by the parent is correct, CHILD2 places its response into the COMMAREA and *then* returns. If the request is not correct the child issues an abend BADC.
- The **EXEC CICS RETURN** command returns to the ASYNCWP program.

*Example 7-17 Program CHILD2 with COMMAREA response*

---

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHILD2.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXEC-RESP PIC S9(8) COMP.
01 REQUEST-TEXT PIC X(30)
 VALUE 'REQUEST FROM PARENT'.
01 RESPONSE-TEXT PIC X(30)
 VALUE 'RESPONSE FROM CHILD2'.

LINKAGE SECTION.
01 DFHCOMMAREA.
 03 REQUEST-AREA PIC X(30).
 03 RESPONSE-AREA PIC X(30).

PROCEDURE DIVISION.

** Main routine *

Main-Routine section.
*
 IF EIBCALEN NOT EQUAL LENGTH OF DFHCOMMAREA

```

```

 PERFORM ABEND-BADC
 END-IF

 IF REQUEST-AREA NOT EQUAL REQUEST-TEXT
 PERFORM ABEND-BADC
 END-IF

 MOVE RESPONSE-TEXT to RESPONSE-AREA

 EXEC CICS RETURN END-EXEC.

Main-Routine-End. EXIT.

```

```

** ABEND-BADC section - this section never returns *

 ABEND-BADC section.
 EXEC CICS ABEND ABCODE('BADC') END-EXEC.
 ABEND-BADC-END. EXIT.

```

---

## 7.3 Tip: Release storage wisely in long-running parent transactions

Asynchronous API applications are usually short lived. CICS automatically cleans up all the asynchronous API state after both participating transactions (parent and child) terminate. CICS persists asynchronous API state to allow a parent transaction to fetch response information from a completed child transaction. This persistence can cause a problem for a long-running parent that creates large numbers of children. CICS persists asynchronous API state for each child of the parent. The parent creates a growing memory footprint where each new child adds to that footprint. A parent transaction can keep this memory footprint under control by using the **FREE CHILD** command to free asynchronous API state for children that it is no longer interested in.

An additional consideration for a long-running parent is cleanup of fetched channels. The **FREE CHILD** command ensures that an unfetched child channel gets deleted. However, if a child channel is fetched, it is owned by the fetching parent program. It is, therefore, the responsibility of that parent program to delete the channel.

Example 7-18 on page 117 shows pseudocode that demonstrates how a long-running parent should use the **FREE CHILD** command to free redundant child state referenced by a child token. This example uses the following information:

- ▶ The logic to generate the REQUEST data and consume the RESPONSE data is omitted in this example.
- ▶ A long-running parent that created collections of children might well use **FETCH ANY** command to fetch the next available child response rather than the **FETCH CHILD** command.
- ▶ In addition to using the **FREE CHILD** command, the parent uses **DELETE CHANNEL** command to delete the fetched channel.

*Example 7-18 Long running parent pseudocode*

---

```
DO FOREVER;

 EXEC CICS PUT CONTAINER('CHLD-REQUEST')
 FROM(REQUEST)
 EXEC CICS RUN TRANSID('CHLD')
 CHANNEL('CHLD-CHANNEL')
 CHILD(CHLD-TOKEN)
 EXEC CICS FETCH CHILD(CHLD-TOKEN)
 COMPSTATUS(COMPSTAT)
 CHANNEL(FETCHED-CHANNEL)
 EXEC CICS GET CONTAINER('CHLD-RESPONSE')
 CHANNEL(FETCHED-CHANNEL)
 INTO(RESPONSE)

 EXEC CICS FREE CHILD(CHLD-TOKEN)
 EXEC CICS DELETE CHANNEL(FETCHED-CHANNEL)
END-DO;
```

---

## 7.4 Trick: Prevent sets of children from interfering in **FETCH ANY** logic by using **FREE CHILD**

This section shows a technique where a parent task creates two collections of children. It then uses the **FREE CHILD** command to free the first collection of children before it creates the second set of children. This process is particularly useful when used in conjunction with the **FETCH ANY** command, where not all child tasks from the first collection are consumed. This method prevents unconsumed child tasks from the first collection from polluting the **FETCH ANY** logic of the second collection.

The following example shows pseudocode of a parent transaction, which has two stages of execution:

- ▶ Example 7-19 shows Stage 1 running three children and fetching the first completed one by using the **FETCH ANY** command.
- ▶ The **FREE CHILD** command then frees the three children from Stage 1.
  - Two of these children have not been fetched, but the parent is no longer interested in their outcome.
  - One child has completed and its response has been fetched. It is safe to free this child, but this is not strictly necessary for Stage 2 to be successful.

*Example 7-19 Stage 1 of PARENT*

---

```
EXEC CICS RUN TRANSID('CHL1')
 CHILD(CHL1-TOK)
 CHANNEL(CHL1-CHAN)
EXEC CICS RUN TRANSID('CHL2')
 CHILD(CHL2-TOK)
 CHANNEL(CHL2-CHAN)
EXEC CICS RUN TRANSID('CHL3')
 CHILD(CHL3-TOK)
 CHANNEL(CHL3-CHAN)
EXEC CICS FETCH ANY(ANY-TOK)
```

```

 COMPSTATUS(CS)
 CHANNEL(ANY-CHAN)
EXEC CICS GET CONTAINER('RESPONSE')
 CHANNEL(ANY-CHAN)
 INTO(RESPONSE)
* IF THE 'RESPONSE' CONTAINER INDICATES 'OK', ENTER STAGE-2 BUT FREE ALL OF THE
STAGE-1 CHILDREN FIRST
EXEC CICS FREE CHILD(CHL1TOK)
EXEC CICS FREE CHILD(CHL2TOK)
EXEC CICS FREE CHILD(CHL3TOK)

```

---

Example 7-20 shows Stage 2 running two children and fetching the first one to complete by using the **FETCH ANY** command. This example fetches a response from one of the Stage 2 children because all the Stage 1 children are free.

*Example 7-20 Stage 2 of PARENT*

---

```

EXEC CICS RUN TRANSID('CHL4')
 CHILD(CHL4-TOK)
 CHANNEL(CHL4-CHAN)
EXEC CICS RUN TRANSID('CHL5')
 CHILD(CHL5-TOK)
 CHANNEL(CHL5-CHAN)
EXEC CICS FETCH ANY(ANY-TOK)
 COMPSTATUS(CS)
 CHANNEL(ANY-CHAN)
EXEC CICS GET CONTAINER('RESPONSE')
 CHANNEL(ANY-CHAN)
 INTO(RESPONSE)
* END OF STAGE-2. EXEC CICS FREE CHILD IS NOT NECESSARY AS PARENT WILL NOW
TERMINATE.

```

---

## 7.5 Tip: Check the status of a child without blocking the parent by using the NOSUSPEND option

By default, the **FETCH CHILD** and **FETCH ANY** commands block until a designated child (**FETCH CHILD**) or an eligible child (**FETCH ANY**) completes. To prevent **FETCH** commands from blocking, you need to use the **NOSUSPEND** option of the **FETCH** commands. You can use this option to just check whether child task has completed because the command returns immediately.

Example 7-21 shows partial code of a parent task using the **NOSUSPEND** option to check if any of its children have completed. It includes the following information:

- ▶ If a child has completed, a routine is called to handle the child response.
- ▶ If no children have completed, the parent executes some of its own business logic.

*Example 7-21 Parent using EXEC CICS FETCH ANY NOSUSPEND*

---

```

EXEC CICS RUN TRANSID('CHL1')
 CHILD(CHL1-TOK)
 CHANNEL(CHL1-CHAN)
EXEC CICS RUN TRANSID('CHL2')
 CHILD(CHL2-TOK)
 CHANNEL(CHL2-CHAN)

```

```

EXEC CICS RUN TRANSID('CHL3')
 CHILD(CHL3-TOK)
 CHANNEL(CHL3-CHAN)
EXEC CICS FETCH ANY(ANY-TOK)
 COMPSTATUS(COMP-STATUS)
 CHANNEL(ANY-CHAN)
 NOSUSPEND RESP(EXEC-RESP)
IF EXEC-RESP = DFHRESP(NORMAL)
 CALL PROCESS-CHILD-RESPONSE
ELSE
 IF EXEC-RESP = DFHRESP(NOTFINISHED)
 CALL PROCESS-PARENT-BUSINESS-LOGIC
 ELSE
 CALL UNEXPECTED-FETCH-ERROR
 END-IF
END-IF

```

---

## 7.6 Trick: Process as many children as possible in a fixed time period

By default, the **FETCH ANY** command blocks until an eligible child completes. It is reasonable for a parent task to wait some time for its children to complete. However, it might not want to wait indefinitely. To prevent an indefinite wait, the **TIMEOUT** option should be used.

If a parent is fetching a series of children by using the **FETCH ANY** command, the parent application might want to reduce the **TIMEOUT** value on each successive **FETCH** command.

Example 7-22 on page 120 shows partial code of a parent task using the **TIMEOUT** option with a loop of **FETCH ANY** commands. It includes the following information:

- ▶ The parent sets a time limit for fetching all its children. In the example, this time limit is called **EXPIRY-TIME**.
- ▶ Before issuing each **FETCH ANY** command, the parent calculates how much time is left to fetch the remaining children. If no time is left, the **FETCH ANY** operation is abandoned and the **TIMED-OUT** flag is set. If there is time left a **FETCH ANY** command is issued using the **TIME-LEFT** value as a **TIMEOUT** time.
- ▶ If a **FETCH ANY** command fails with **NOTFINISHED** it means the operation has timed out and the **TIMED-OUT** flag is set.
- ▶ If the **FETCH ANY** command fails with a **NOTFND** condition, it indicates that all the children have been fetched.
- ▶ If the **FETCH ANY** command completes normally a section called **MATCH-CHILD** is called to identify which child has been fetched and process the response.
- ▶ **MATCH-CHILD** code is not shown here because this example focuses on the use of **TIMEOUT**.
- ▶ In this example, **ABSTIME** values such as **START-TIME**, **EXPIRY-TIME** and **CURRENT-TIME** are 8-byte packed decimal fields. The **TIME-LEFT** field is a 4-byte binary field.

Figure 7-3 shows a pictorial representation of this scenario. In this figure, the PARENT program fetches the responses of child CHL1 and child CHL3. Child CHL2 is not fetched because it takes too long to complete.

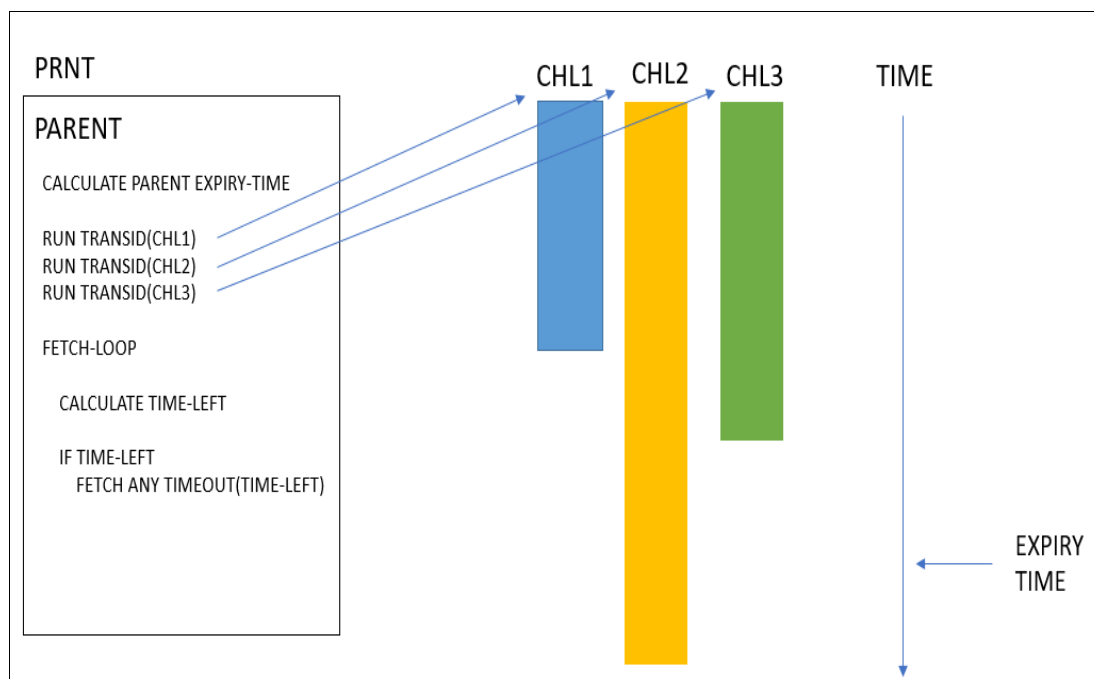


Figure 7-3 Parent using *FETCH ANY* loop with *TIMEOUT*

*Example 7-22 Parent using EXEC CICS FETCH ANY TIMEOUT()*

Main-Routine section.

\*

```
EXEC CICS ASKTIME ABSTIME(START-TIME) END-EXEC
COMPUTE EXPIRY-TIME = (START-TIME + TIME-LIMIT)
EXEC CICS RUN TRANSID('CHL1')
 CHILD(CHL1-TOK)
 CHANNEL(CHL1-CHAN)
EXEC CICS RUN TRANSID('CHL2')
 CHILD(CHL2-TOK)
 CHANNEL(CHL2-CHAN)
EXEC CICS RUN TRANSID('CHL3')
 CHILD(CHL3-TOK)
 CHANNEL(CHL3-CHAN)
MOVE 'N' to TIMED-OUT
PERFORM FETCH-ANY UNTIL
 (EXEC-RESP EQUAL DFHRESP(NOTFND) OR
 TIMED-OUT EQUAL 'Y')
```

\* TERMINATE PARENT program

FETCH-ANY section.

\*

```
EXEC CICS ASKTIME ABSTIME(CURRENT-TIME) END-EXEC

IF CURRENT-TIME < EXPIRY-TIME
 COMPUTE TIME-LEFT = (EXPIRY-TIME - CURRENT-TIME)
ELSE
```

```

 MOVE 'Y' TO TIMED-OUT
 GO TO FETCH-ANY-END
 END-IF
 EXEC CICS FETCH ANY(FETCH-ANY-TOKEN)
 CHANNEL(FETCH-ANY-CHANNEL)
 COMPSTATUS(FETCH-ANY-COMPSTATUS)
 ABCODE(FETCH-ANY-ABCODE)
 TIMEOUT(TIME-LEFT)
 RESP(EXEC-RESP) END-EXEC

 IF EXEC-RESP EQUAL DFHRESP(NORMAL)
 PERFORM MATCH-CHILD
 ELSE
 IF EXEC-RESP EQUAL DFHRESP(NOTFINISHED)
 MOVE 'Y' TO TIMED-OUT
 ELSE
 IF EXEC-RESP NOT EQUAL DFHRESP(NOTFND)
 PERFORM ABEND-BADC
 END-IF
 END-IF
 END-IF.
 FETCH-ANY-END.
 EXIT.

```

---

## 7.7 Tip: Using response-only channels between parent and child transactions

This next example shows how to use a response-only channel with the asynchronous API. In this example, a child transaction performs a function which is pre-designated to open a list of files. The parent doesn't need to provide any request information, but it is interested in the results of running the child. In this case, the parent passes an empty channel when starting the child transaction with the **RUN TRANSID** command so that the child can return a response.

Example 7-23 on page 122 shows a parent program running a child transaction called **FILE0**. An empty channel called 'FILE-CHANNEL' is passed to the child. The response channel is fetched from the completed child as **FETCHED-CHAN**. The parent **GETs** a **FAILED-FILES** container from this channel and reports any files, which could not be opened.

Figure 7-4 on page 122 shows a pictorial representation of this scenario. In this figure, the **FILE0032** and **FILE0088** files fail to open.

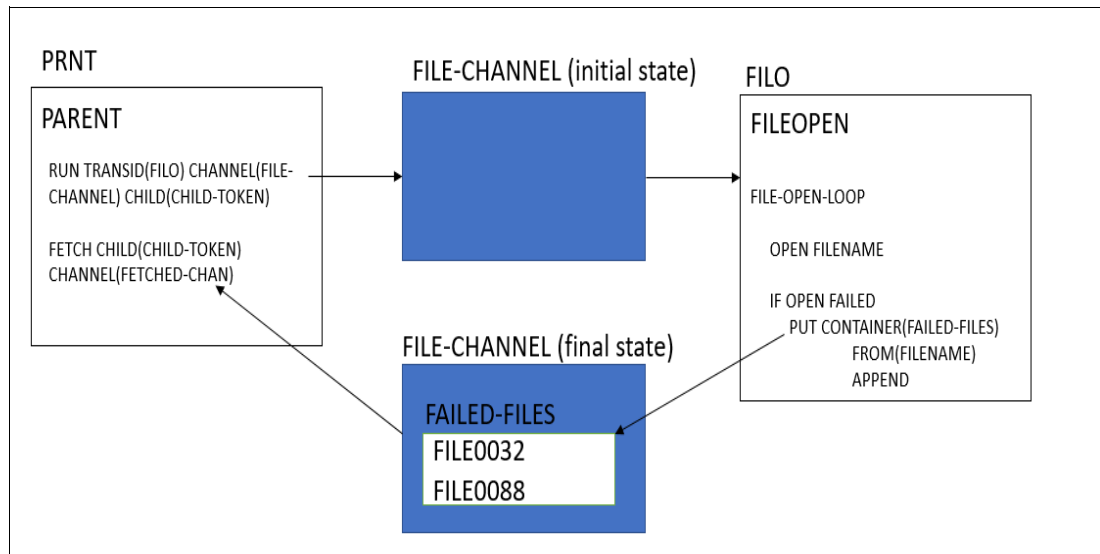


Figure 7-4 Parent PRNT passing an empty channel to child FILO

*Example 7-23 Parent program using response only channel*

```

EXEC CICS RUN TRANSID('FILO')
 CHILD(CHILD-TOKEN)
 CHANNEL('FILE-CHANNEL')
 RESP(EXEC-RESP)
IF EXEC-RESP = DFHRESP(NORMAL)
 EXEC CICS FETCH CHILD(CHILD-TOKEN)
 CHANNEL(FETCHED-CHAN)
 COMPSTATUS(COMP-STATUS)
 RESP(EXEC-RESP)
 IF (EXEC-RESP = DFHRESP(NORMAL)) AND
 (COMP-STATUS = DFHVALUE(NORMAL))
 EXEC CICS GET CONTAINER('FAILED-FILES')
 CHANNEL(FETCHED-CHAN)
 SET() FLENGTH()
 RESP(EXEC-RESP)
 IF EXEC-RESP = DFHRESP(NORMAL)
 CALL REPORT-FAILED-FILES
 END-IF
 END-IF
END-IF

```

Example 7-24 shows the child program opening a list of files. If any file fails to open, the name of the file is appended to the list in the 'FAILED-FILES' container.

*Example 7-24 Child program opening a list of files*

```

FILE-OPEN-LOOP.
 EXEC CICS OPEN FILE(NEXT-FILE) RESP(EXEC-RESP)
 IF EXEC-RESP NOT EQUAL DFHRESP(NORMAL)
 EXEC CICS PUT CONTAINER('FAILED-FILES') APPEND
 FROM(NEXT-FILE)
 END-IF
FILELIST
FILE0001

```



FILE0002  
FILE0003  
...  
FILE0099  
**ENDLIST**

---





## Debugging and problem determination

This chapter provides guidance about debugging asynchronous API applications and provides information about system management techniques. It includes the following sections:

- ▶ Using the CICS execution diagnostic facility: CEDF and CEDX
- ▶ Asynchronous APIabend code
- ▶ Tracing asynchronous API applications
- ▶ Sample application trace flow using FETCH ANY commands
- ▶ Sample application trace flow using FETCH CHILD commands and the NOSUSPEND and TIMEOUT options
- ▶ Sample application trace flow using FREE CHILD commands
- ▶ Transaction dumps and the asynchronous API
- ▶ System dumps and the asynchronous API

## 8.1 Using the CICS execution diagnostic facility: CEDF and CEDX

If an asynchronous parent transaction can be run from a terminal, you can use the CICS-supplied transaction, CEDF, to step through the program execution. CEDF displays each CICS command and the program initiation and termination.

Asynchronous child transactions always run as *non-terminal transactions*. Thus, you must use another CICS-supplied transaction, CEDX, to debug asynchronous child tasks. You must also use CEDX to debug an asynchronous parent if it runs as nonterminal.

You can find more information about using CEDF and CEDX in [IBM Knowledge Center](#).

You can use CEDX to simulate delays in child transactions and to force child transactions to abend. This process can help you to test how asynchronous parent tasks deal with incomplete or failed child tasks. You can also use CEDX to force children to complete in certain orders.

The following example uses CEDF to debug an asynchronous parent and uses CEDX to debug the two asynchronous children that were created by the parent. In this example, PRNT is the parent transaction that runs the PARENT program at a terminal. The CHL1 and CHL2 transactions are the child transactions that run the CHILD1 and CHILD2 programs respectively.

The assembler source code for the PARENT (Figure 8-1), CHILD1 (Figure 8-2 on page 127), and CHILD2 (Figure 8-3 on page 127) programs is shown in the figures that follow.

```
DFHEISTG DSECT
CHL1TOK DS CL16
CHL2TOK DS CL16
FETCHTOK DS CL16
RESPCHAN DS CL16
COMPSTAT DS CL4
ABCODE DS CL4
PARENT CSECT
PARENT AMODE 31
PARENT RMODE ANY
EXEC CICS RUN TRANSID('CHL1') CHANNEL('CHAN') CHILD(CHL1TOK)
EXEC CICS RUN TRANSID('CHL2') CHANNEL('CHAN') CHILD(CHL2TOK)
EXEC CICS FETCH ANY(FETCHTOK) CHANNEL(RESPCHAN) X
 COMPSTATUS(COMPSTAT) ABCODE(ABCODE)
CLC FETCHTOK,CHL1TOK
BNE TESTCHL2
EXEC CICS SEND TEXT FROM(CHL1FTCH) ERASE FREEKB
B GETNEXT
TESTCHL2 DS 0H
CLC FETCHTOK,CHL2TOK
BNE ABEND
EXEC CICS SEND TEXT FROM(CHL2FTCH) ERASE FREEKB
GETNEXT DS 0H
EXEC CICS FETCH ANY(FETCHTOK) CHANNEL(RESPCHAN) X
 COMPSTATUS(COMPSTAT) ABCODE(ABCODE)
EXEC CICS RETURN
ABEND DS 0H
EXEC CICS ABEND ABCODE('BADC')
CHL1FTCH DC CL30'CHL1 COMPLETED FIRST'
CHL2FTCH DC CL30'CHL2 COMPLETED FIRST'
END PARENT
```

Figure 8-1 The PARENT program

```

CHILD1 CSECT
CHILD1 AMODE 31
CHILD1 RMODE ANY
 EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESPI)
 EXEC CICS RETURN
RESP DC CL30'NORMAL RESPONSE FROM CHL1'
 END CHILD1

```

Figure 8-2 The CHILD1 program

```

CHILD2 CSECT
CHILD2 AMODE 31
CHILD2 RMODE ANY
 EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESPI)
 EXEC CICS RETURN
RESP DC CL30'NORMAL RESPONSE FROM CHL2'
 END CHILD2

```

Figure 8-3 The CHILD2 program

To complete this example, follow these steps:

1. On terminal 1, enter CEDF, and then type PRNT to run the parent transaction.
2. On terminal 2, enter CEDX CHL1, and on terminal 3, enter CEDX CHL2.

Terminal 1 displays the information shown in Figure 8-4.

```

TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: PROGRAM INITIATION

 EIBTIME = 110737
 EIBDATE = 0117254
 EIBTRNID = 'PRNT'
 EIBTASKN = 200
 EIBTRMID = 'TC40'

 EIBCPOSN = 4
 EIBCALEN = 0
 EIBRID = X'7D'
 EIBFN = X'0000'
 EIBRCODE = X'000000000000'
 EIBDS = '.....'
+ EIBREQID = '.....'

 AT X'2AE0011A'
 AT X'2AE0011B'
 AT X'2AE0011D'

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-4 The PARENT program starting

- Continue to press Enter on terminal 1 until the first EXEC CICS RUN TRANSID command completes (Figure 8-5).

```

TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS RUN TRANSID
TRANSID ('CHL1')
CHANNEL ('CHAN')
CHILD ('...&.....')

OFFSET:X'000190' LINE: EIBFN=X'343E'
RESPONSE: NORMAL EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-5 The first RUN TRANSID command issued by the PARENT program

- Now, switch to terminal 2. The information shown in Figure 8-6 indicates that the CHL1 task has started.

```

TRANSACTION: CHL1 PROGRAM: CHILD1 TASK: 0000203 APPLID: IY2CZCCM DISPLAY: 00
STATUS: PROGRAM INITIATION

EIBTIME = 110829
EIBDATE = 0117254
EIBTRNID = 'CHL1'
EIBTASKN = 203
EIBTRMID = '....'

EIBCPOSN = 0
EIBCALEN = 0
EIBRID = X'00' AT X'2C10011A'
EIBFN = X'0000' AT X'2C10011B'
EIBRCODE = X'000000000000' AT X'2C10011D'
EIBDS = '.....'
+ EIBREQID = '.....'

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-6 The CHL1 task starts

- Back on terminal 1, continue to press Enter until the second EXEC CICS RUN TRANSID command completes (Figure 8-7).

```

TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS RUN TRANSID
TRANSID ('CHL2')
CHANNEL ('CHAN')
CHILD ('...&...Q.....')

OFFSET:X'0001E2' LINE: EIBFN=X'343E'
RESPONSE: NORMAL EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: ABEND USER TASK

```

Figure 8-7 The second RUN TRANSID command issued by the PARENT program

- Now switch to terminal 3. The information shown in Figure 8-8 indicates that the CHL2 task has started.

```

TRANSACTION: CHL2 PROGRAM: CHILD2 TASK: 0000208 APPLID: IY2CZCCM DISPLAY: 00
STATUS: PROGRAM INITIATION

EIBTIME = 110910
EIBDATE = 0117254
EIBTRNID = 'CHL2'
EIBTASKN = 208
EIBTRMID = '....'

EIBCPOSN = 0
EIBCALEN = 0
EIBRID = X'00' AT X'2C20011A'
EIBFN = X'0000' AT X'2C20011B'
EIBRCODE = X'000000000000' AT X'2C20011D'
EIBDS = '.....'
+ EIBREQID = '.....'

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-8 The CHL2 task starts

7. Continue to press Enter on terminal 3 until the task termination screen for the CHL2 task displays, as shown in Figure 8-9.

```

TRANSACTION: CHL2 TASK: 0000208 APPLID: IY2CZCCM DISPLAY: 00
STATUS: TASK TERMINATION

CONTINUE EDF? (ENTER YES OR NO) REPLY: YES
ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-9 The CHL2 task terminates

8. Press PF3 to allow the CHL2 task to complete normally. At this point the CHL2 task has finished, but the CHL1 task is still active.
9. Switch back to terminal 1, and press Enter until the first **EXEC CICS FETCH ANY** command is completed by the parent task (Figure 8-10).

```

TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS FETCH ANY
ANY ('...&...Q.....')
CHANNEL ('DFHAS00002-00208')
COMPSTATUS (1016)
ABCODE (' ')

OFFSET:X'00023A' LINE: EIBFN=X'3444'
RESPONSE: NORMAL EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: ABEND USER TASK

```

Figure 8-10 The first **FETCH ANY** command issued by the **PARENT** program



The following information displays as the response from the CHL2 task (the second child created by the parent):

- The name prefix of the fetched CHANNEL is 'DFHAS00002'. CICS creates this channel name as part of the **FETCH** command. You might see that the child sequence number is included after the DFHAS prefix. The last 5-digits of the channel name matches the child task number.
- The COMPSTATUS of 1016 is the CVDA value for NORMAL.
- The parent application compares the fetched child token (returned in the ANY option) against the two tokens that were created by the earlier **EXEC CICS RUN TRANSID** commands.

The parent task issues an appropriate message to the terminal user as shown in Figure 8-11.

```
TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS SEND TEXT
FROM ('CHL2 COMPLETED FIRST ')
LENGTH (30)
TERMINAL
FREEKB
ERASE

OFFSET:X'00029A' LINE: EIBFN=X'1806'
RESPONSE: NORMAL EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: ABEND USER TASK
```

*Figure 8-11 Message to the terminal user issued by the PARENT program*

10. Now return to terminal 2. Step to the **EXEC CICS PUT CONTAINER** command issued by the CHL1 task. Press PF12, and enter an abend code of XXXX in the REPLY field (Figure 8-12).

```

TRANSACTION: CHL1 PROGRAM: CHILD1 TASK: 0000203 APPLID: IY2CZCCM DISPLAY: 00
STATUS: ABOUT TO EXECUTE COMMAND
EXEC CICS PUT CONTAINER
CONTAINER ('RESPONSE ')
FROM ('NORMAL RESPONSE FROM CHL1 ')
FLENGTH (30)

OFFSET=X'000166' LINE: EIBFN=X'3416'

ENTER ABEND CODE AND REQUEST ABEND AGAIN REPLY: XXXX
ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: ABEND USER TASK

```

Figure 8-12 Trigger abend code XXXX against the CHL1 task

11. Press PF12 again to trigger an abend against the CHL1 task (Figure 8-13).

```

TRANSACTION: CHL1 PROGRAM: DFHEDFX TASK: 0000203 APPLID: IY2CZCCM DISPLAY: 00
STATUS: AN ABEND HAS OCCURRED

EIBTIME = 110829
EIBDATE = 0117254
EIBTRNID = 'CHL1'
EIBTASKN = 203
EIBTRMID = '....'

EIBCPOSN = 0
EIBCALEN = 0
EIBRID = X'00'
EIBFN = X'3416' PUT
EIBRCODE = X'000000000000'
EIBDS = '.....'
+ EIBREQID = '.....'

ABEND : XXXX

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: UNDEFINED

```

Figure 8-13 Abend code XXXX triggered

12. Press PF3 on terminal 2 to allow the CHL1 task to terminate abnormally with the abend code of XXXX.

13.Back on terminal 1, step to the last **EXEC CICS FETCH ANY** command (Figure 8-14).

```
TRANSACTION: PRNT PROGRAM: PARENT TASK: 0000200 APPLID: IY2CZCCM DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS FETCH ANY
ANY ('...&.....')
CHANNEL ('DFHAS00001-00203')
COMPSTATUS (900)
ABCODE ('XXXX')

OFFSET:X'0002F2' LINE: EIBFN=X'3444'
RESPONSE: NORMAL EIBRESP=0

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : SWITCH HEX/CHAR PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: EIB DISPLAY PF12: ABEND USER TASK
```

Figure 8-14 The last **FETCH ANY** command issued by the **PARENT** program

The **COMPSTATUS** of 900 is the **CVDA** value for the **ABEND** command. You can also see the **XXXX** abend code in the **ABCODE** field.

## 8.2 Asynchronous API abend code

You can encounter abend code **AASA** when an asynchronous child starts running and then an unexpected error occurs when performing non-terminal sign on for the child's **USERID**. The child inherits the **USERID** from the parent task. This abend can occur if the parent **USERID** is revoked before the child task executed.

## 8.3 Tracing asynchronous API applications

Many application problems that arise from using the asynchronous API can be solved by obtaining a *CICS trace*. A CICS trace can be directed to the CICS auxiliary trace data sets or to a GTF trace data set.

CICS trace also appears in a transaction dump and a system dump, providing that CICS internal tracing is active and that the master system trace flag is on.

Trace points relating to asynchronous API are controlled by the component trace points. See Table 8-1.

Table 8-1 Key trace components for asynchronous API tracing

Trace component	Effect
Asynchronous services domain (AS) level-1	Basic tracing of asynchronous services domain activity
AS level-2	Extended tracing of asynchronous services domain activity
Exec interface (EI) level-1	Basic tracing of EXEC layer (EIP and EIBAM tracing)

**Selecting trace components:** You can select trace components by enabling the **Components** option of the CICS supplied transaction CETR.

## 8.4 Sample application trace flow using FETCH ANY commands

This section shows selected trace output of a sample asynchronous API application that we ran where the parent used the **EXEC CICS FETCH ANY** command. The trace was directed to a CICS auxiliary trace data set and was formatted using a CICS supplied, auxiliary trace formatter. All the trace entries shown were formatted with the **ABBREV** keyword unless otherwise stated.

### 8.4.1 The environment

For this example, we defined a parent transaction (**PRNT**) that used an assembler program called **PARENT**, as shown in Figure 8-15 on page 135 and Figure 8-16 on page 136. The parent used the **EXEC CICS RUN TRANSID** command to create the following child tasks:

- ▶ **CHL1**
- ▶ **CHL2**

We defined the **PRNT** transaction with **RESec:Yes**. The CICS region had security active and had an **XPCT** resource class active. Thus, the **RUN TRANSID** commands that were issued by the **PARENT** program were subject to security checking.

The **PARENT** program was defined with **Concurrency:Threadsafe** and **Api:Openapi**. These setting meant that the **PARENT** program ran on an open task control block (TCB).

We also had two child transactions, called **CHL1** and **CHL2**. **CHL1** ran the assembler program **CHIL1**, and **CHL2** ran the assembler program **CHIL2**. These transactions were all defined with default transaction and program attributes. The code for the **CHIL1** and **CHIL2** programs is shown in Figure 8-17 on page 137 and Figure 8-18 on page 137.

**Important:** The **PARENT** program includes a deliberate bug that causes the **CHL1** child transaction to abend.

DFHEISTG DSECT			
CHL1TOK	DS	CL16	CHILD TOKEN FOR CHL1
CHL2TOK	DS	CL16	CHILD TOKEN FOR CHL2
FETCHTOK	DS	CL16	CHILD TOKEN USED ON FETCH ANY COMMAND
FETCHRESPCHAN	DS	CL16	RESPONSE CHANNEL ON FETCH ANY COMMAND
ABCODE	DS	CL4	ABEND CODE ON FETCH COMMAND
COMPSTAT	DS	CL4	COMPLETION STATUS ON FETCH COMMAND
FLEN	DS	F	FULLWORD LENGTH
CHL1RESP	DS	CL30	CONTAINER RESPONSE FROM CHL1
CHL2RESP	DS	CL30	CONTAINER RESPONSE FROM CHL2

Figure 8-15 Working storage for the PARENT program

The PARENT program's code includes the following details (as shown in Figure 8-16 on page 136):

- ▶ The PARENT program issues two **EXEC CICS RUN TRANSID** commands to create two children and passes a populated channel to each child.
- ▶ It then repeats the **EXEC CICS FETCH ANY** commands to fetch all the responses from its children. This command also fetches the response channel or channels from the children.  
A NOTFND condition on the **EXEC CICS FETCH ANY** command indicates that all children have been fetched.
- ▶ If the CHL1 task completes normally, the PARENT program extracts the RESPONSE container from the CHL1 task's channel. It then sends the following message to the terminal:  
CHL1 CHILD SUCCESSFUL
- ▶ If the CHL1 task fails in some way (for example, if it abends or fails to return a RESPONSE container), the PARENT program sends the following message to the terminal:  
CHL1 CHILD FAILED

```

PARENT CSECT
PARENT AMODE 31
PARENT RMODE ANY
RUNCHL1 DS 0H
EXEC CICS PUT CONTAINER('REQUESTER') FROM(REQ1) *
CHANNEL('CHL1-CHANNEL')
EXEC CICS RUN TRANSID('CHL1') CHILD(CHL1TOK) *
CHANNEL('CHL1-CHANNEL')
RUNCHL2 DS 0H
EXEC CICS PUT CONTAINER('REQUEST') FROM(REQ2) *
CHANNEL('CHL2-CHANNEL')
EXEC CICS RUN TRANSID('CHL2') CHILD(CHL2TOK) *
CHANNEL('CHL2-CHANNEL')
FETCHNXT DS 0H
EXEC CICS FETCH ANY(FETCHTOK) CHANNEL(FETCHRESPCHAN) *
COMPSTATUS(COMPSTAT) ABCODE(ABCODE) NOHANDLE
CLC EIBRESP,DFHRESP(NOTFND) ALL CHILDREN FETCHED ?
BE ALLDONE YES, GOTO ALLDONE
CLC FETCHTOK,CHL1TOK CHL1 FETCHED ?
BNE TRYCHL2 NO, GO TRY CHL2
CLC COMPSTAT,DFHVALUE(NORMAL) CHL1 COMPLETED NORMALLY ?
BNE CHL1ERR NO, GOTO ERROR HANDLING
MVC FLEN,=F'30' LENGTH OF RESPONSE CHANNEL
EXEC CICS GET CONTAINER('RESPONSE') CHANNEL(FETCHRESPCHAN) *
INTO(CHL1RESP) FLENGTH(FLEN)
EXEC CICS SEND TEXT FROM(CHL1GOOD) FREEKB ERASE
B FETCHNXT
TRYCHL2 DS 0H
CLC FETCHTOK,CHL2TOK CHL2 FETCHED ?
BNE ABEND NO, THIS SHOULDN'T HAPPEN !
*** WE COULD CHECK CHL2 RESPONSE HERE !
B FETCHNXT
NORMRET DS 0H
EXEC CICS RETURN
CHL1ERR DS 0H
EXEC CICS SEND TEXT FROM(CHL1BAD) FREEKB ERASE
B FETCHNXT
ALLDONE DS 0H
EXEC CICS RETURN
ABEND DS 0H
EXEC CICS ABEND ABCODE('BADF')
REQ1 DC CL30'PARENT REQUEST FOR CHILD1'
REQ2 DC CL30'PARENT REQUEST FOR CHILD2'
CHL1GOOD DC CL30'CHL1 CHILD SUCCESSFUL'
CHL1BAD DC CL30'CHL1 CHILD FAILED'
END PARENT

```

Figure 8-16 Application code for the PARENT program

Figure 8-17 on page 137 and Figure 8-18 on page 137 show the code for the CHILD1 and CHILD2 assembler programs. These programs are almost identical. Both expect to receive a container called REQUEST. If this container isn't present, an abend is issued (abend code BADC). Otherwise, a RESPONSE container is placed onto the channel. This container is fetched later by the parent.

```

DFHEISTG DSECT
FLEN DS F
CONTAREA DS CL30
CHILD1 CSECT
CHILD1 AMODE 31
CHILD1 RMODE ANY
 MVC FLEN,=F'30'
 EXEC CICS GET CONTAINER('REQUEST') INTO(CONTAREA) *
 FLENGTH(FLEN) NOHANDLE
 CLC EIBRESP,DFHRESP(NORMAL)
 BNE ABEND
 CLC CONTAREA,REQ1
 BNE ABEND
 EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESF)
RETURN DS 0H
 EXEC CICS RETURN
ABEND DS 0H
 EXEC CICS ABEND ABCODE('BADC')
REQ1 DC CL30'PARENT REQUEST FOR CHILD1'
RESF DC CL30'NORMAL RESPONSE FROM CHL1'
END CHILD1

```

Figure 8-17 Code for the CHILD1 program

```

DFHEISTG DSECT
FLEN DS F
CONTAREA DS CL30
CHILD2 CSECT
CHILD2 AMODE 31
CHILD2 RMODE ANY
 EXEC CICS DELAY INTERVAL(000004)
 MVC FLEN,=F'30'
 EXEC CICS GET CONTAINER('REQUEST') INTO(CONTAREA) *
 FLENGTH(FLEN) NOHANDLE
 CLC EIBRESP,DFHRESP(NORMAL)
 BNE ABEND
 CLC CONTAREA,REQ2
 BNE ABEND
 EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESF)
RETURN DS 0H
 EXEC CICS RETURN
ABEND DS 0H
 EXEC CICS ABEND ABCODE('BADC')
REQ2 DC CL30'PARENT REQUEST FOR CHILD2'
RESF DC CL30'NORMAL RESPONSE FROM CHL2'
END CHILD2

```

Figure 8-18 Code for CHILD2 program

## 8.4.2 Trace of the PARENT program creating two children

When tracing a PARENT program, keep in mind the following considerations:

- ▶ The first thing the PARENT application does is to populate a request channel with a request container. It then runs the first child task, passing the request channel.
- ▶ The PARENT program then repeats this process, running a second child task and passing a second request channel.
- ▶ The PARENT program runs on an open TCB - L9000 in the trace. This process allows the PRNT task to run in parallel with its child tasks. In this example, both children both run on the QR TCB.
- ▶ The PRNT program runs as task 00059.

Figure 8-19 shows the PARENT program creating channel CHL1-CHANNEL and populating it with a container called REQUESTER.

```

00059 L9000 AP 00E1 EIP ENTRY PUT-CONTAINER 0004,2AE008A0 .\...,09003416,0440 =000228=
00059 L9000 AP F801 EIBAM ENTRY PUT_CONTAINER =000229=
00059 L9000 PG 1700 PGCH ENTRY INQUIRE_CHANNEL CHL1-CHANNEL =000230=
00059 L9000 PG 1701 PGCH EXIT INQUIRE_CHANNEL/EXCEPTION CHANNEL_NOT_FOUND,00000000 =000231=
00059 L9000 PG 1700 PGCH ENTRY CREATE_CHANNEL CHL1-CHANNEL,CURRENT,NO =000232=
00059 L9000 SM 0301 SMGF ENTRY GETMAIN 48E4041C , 0000006F,40,YES,00,CHCB,YES =000233=
00059 L9000 SM 0302 SMGF EXIT GETMAIN/OK 2AF6D030 =000234=
00059 L9000 PG 1800 PGCP ENTRY CREATE_CONTAINER_POOL 25,NO,YES =000235=
00059 L9000 SM 0301 SMGF ENTRY GETMAIN 48E404E8 , 00000070,30,YES,00,CPCB,YES =000236=
00059 L9000 SM 0302 SMGF EXIT GETMAIN/OK 2AF78030 =000237=
00059 L9000 PG 1801 PGCP EXIT CREATE_CONTAINER_POOL/OK 2AF78030 =000238=
00059 L9000 PG 1701 PGCH EXIT CREATE_CHANNEL/OK 2AF6D030,2AF78030 =000239=
00059 L9000 PG 1900 PGCR ENTRY PUT_CONTAINER 2AF78030,REQUESTER,EXEC,2C00142E , 0000001E =000240=
00059 L9000 SM 4201 S2GF ENTRY GETMAIN 00000050_40904AE8 , 00000000_00000074,1000,YES,CSDB,YES,2AC67880 =000241=
00059 L9000 SM 4202 S2GF EXIT GETMAIN/OK 00000050_41000000 =000242=
00059 L9000 PG 1901 PGCR EXIT PUT_CONTAINER/OK 2AF79030,1,1 =000243=
00059 L9000 AP F802 EIBAM EXIT PUT_CONTAINER RESP=0 RESP2=0 =000244=
00059 L9000 AP 00E1 EIP EXIT PUT-CONTAINER OK 00F4,00000000,00003416,0240 =000245=

```

Figure 8-19 Parent creating channel for CHL1 child task

Figure 8-20 shows the first child task being created. The process is as follows:

- ▶ There is a security check to authorize that the current USERID (CICSUSER) has authority to start the CHL1 task.
- ▶ The (CHL1-CHANNEL) passed channel is then copied, and DFHASAS is called to start CHL1 (trace =000270=).
- ▶ DFHASAS saves the security context of the parent with a FLATTEN\_TRANSACTION\_USER call (=000272=). It then attaches the CHL1 task.

```

00059 L9000 AP 00E1 EIP ENTRY RUN-TRANSID 0004,2AE008A0 .\...,0900343E,0440 =000246=
00059 L9000 AP F801 EIBAM ENTRY RUN_TRANSID =000247=
00059 L9000 XS 0701 XSRC ENTRY CHECK_CICS_RESOURCE CHL1,TRANSACTION,READ =000248=
00059 L9000 XS 0709 XSRC EVENT CHECK CHL1 CHECK_RESOURCE_ACCESS,29956530 , 00000001,AFV01PCT,READ,YES,2AC6 =000249=
00059 L9000 XS 070A XSRC EVENT CHECK-COMplete CHL1 CICSUSER CHECK_RESOURCE_ACCESS,OK,0,0,0,0 =000250=
00059 L9000 XS 0702 XSRC EXIT CHECK_CICS_RESOURCE/OK =000251=
00059 L9000 PG 1700 PGCH ENTRY INQUIRE_CHANNEL CHL1-CHANNEL =000252=
00059 L9000 PG 1701 PGCH EXIT INQUIRE_CHANNEL/OK 2AF6D030,2AF78030 =000253=
00059 L9000 PG 1700 PGCH ENTRY COPY_CHANNEL 2AF6D030,YES =000254=
00059 L9000 SM 0301 SMGF ENTRY GETMAIN 48E4041C , 0000006F,40,YES,00,CHCB,YES =000255=
00059 L9000 SM 0302 SMGF EXIT GETMAIN/OK 2AF6D0B0 =000256=
00059 L9000 PG 1800 PGCP ENTRY COPY_CONTAINER_POOL 2AF78030,YES =000257=
00059 L9000 SM 0301 SMGF ENTRY GETMAIN 48E404E8 , 00000070,30,YES,00,CPCB,YES =000258=
00059 L9000 SM 0302 SMGF EXIT GETMAIN/OK 2AF78090 =000259=
00059 L9000 PG 1900 PGCR ENTRY COPY_CONTAINER 2AF79030,2AF78090 =000260=
00059 L9000 SM 4201 S2GF ENTRY GETMAIN 00000050_40904AE8 , 00000000_00000074,1000,YES,CSDB,YES,2AC683C0 =000261=
00059 L9000 SM 4202 S2GF EXIT GETMAIN/OK 00000050_41001000 =000262=
00059 L9000 PG 1901 PGCR EXIT COPY_CONTAINER/OK =000263=
00059 L9000 PG 1801 PGCP EXIT COPY_CONTAINER_POOL/OK 2AF78090 =000264=
00059 L9000 PG 1701 PGCH EXIT COPY_CHANNEL/OK 2AF6D0B0 =000265=
00059 L9000 XM 0401 XMLD ENTRY LOCATE_AND_LOCK_TRANDEF CHL1,YES =000266=
00059 L9000 DD 0301 DDLO ENTRY LOCATE 29A00060,2AC67314,TXD,CHL1 =000267=
00059 L9000 DD 0302 DDLO EXIT LOCATE/OK 2ADC0C90 , D7000000 =000268=
00059 L9000 XM 0402 XMLD EXIT LOCATE_AND_LOCK_TRANDEF/OK 2ADC1D98 , 000000B4 =000269=
00059 L9000 AS 0300 ASAS ENTRY RUN_TRANSACTION CHL1,2AF6D0B0 =000270=
00059 L9000 AS 0305 ASAS EVENT ACB_INITIALIZED =000271=
00059 L9000 US 0401 USXM ENTRY FLATTEN_TRANSACTION_USER 2AF7B0BC , 00000000 , 00000030 =000272=
00059 L9000 US 0402 USXM EXIT FLATTEN_TRANSACTION_USER/OK 2AF7B0BC , 00000000 , 00000030 =000273=
00059 L9000 XM 1101 XMAT ENTRY ATTACH CHL1,NONE,C,YES,YES,AS_RUN_TRANSACTION,2AF7B090 , 00000060,SAME,YES =000274=
00059 L9000 XM 0401 XMLD ENTRY LOCATE_AND_LOCK_TRANDEF CHL1,AVAILABLE_ONLY =000275=
00059 L9000 DD 0301 DDLO ENTRY LOCATE 29A00060,2AC68B0C,TXD,CHL1 =000276=
00059 L9000 DD 0302 DDLO EXIT LOCATE/OK 2ADC0C90 , D7000000 =000277=
00059 L9000 XM 0402 XMLD EXIT LOCATE_AND_LOCK_TRANDEF/OK 2ADC1D98 , 000000B4,CHL1 =000278=
00059 L9000 MN 0B0A MNOD ENTRY GET_ODR 2AC688E8 , 00000000 , 00000200,YES =000279=
00059 L9000 MN 0B0B MNOD EXIT GET_ODR/OK 2AC688E8 , 00000200 , 00000200,0 =000280=
00059 L9000 SM 0301 SMGF ENTRY GETMAIN 299435B4 , 0000000A,210,YES,XMATODR =000281=
00059 L9000 SM 0302 SMGF EXIT GETMAIN/OK 29A05CA0 =000282=
00059 L9000 MN 0D01 MNAC ENTRY GET_ACD 2AC68324 , 00000000 , 000000CC,2AC683F0 , 00000000 , 000000CC =000283=
00059 L9000 MN 0D02 MNAC EXIT GET_ACD/OK 2AC68324 , 00000000 , 000000CC,2AC683F0 , 00000000 , 000000CC,NO,NO,N =000284=
00059 L9000 DS 0002 DSAT ENTRY ATTACH 2ACDE700,0,1,NON_SYSTEM,2ACDE700 , 0000060C =000285=
00059 L9000 DS 0003 DSAT EXIT ATTACH/OK 04860005 =000286=
00059 L9000 XM 1102 XMAT EXIT ATTACH/OK 0000060C =000287=
00059 L9000 AS 0301 ASAS EXIT RUN_TRANSACTION/OK 00000050_418000D8 , 0000060C_00000001 =000288=
00059 L9000 AP F802 EIBAM EXIT RUN_TRANSID RESP=0 RESP2=0 =000289=
00059 L9000 AP 00E1 EIP EXIT RUN-TRANSID OK 00F4,00000000,0000343E,0240 =000290=

```

Figure 8-20 Parent issuing RUN TRANSID command to start CHL1 child task



### 8.4.3 Trace of one child

Figure 8-21 shows the CHL1 child task starting up. The CHL1 task runs as task number 00060 and runs in parallel with its parent, but it filters out the trace entries that are created by the PRNT task.

- ▶ DFHASXM INIT\_XM\_CLIENT (trace =000350=) shows the child task establishing its security context. You can see it using the parent's USERID (CICSUSER) (trace =000355=).
- ▶ DFHASXM BIND\_XM\_CLIENT (trace =000410=) binds the channel that is passed by the parent. This channel becomes the current channel when the CHIL1 program runs.

00060	QR	AS	0200	ASXM	ENTRY	INIT_XM_CLIENT	2AF7B090 , 00000060	=000350=
00060	QR	XM	1001	XMIQ	ENTRY	SET_TRANSACTION	NO	=000351=
00060	QR	XM	1002	XMIQ	EXIT	SET_TRANSACTION/OK		=000352=
00060	QR	US	0401	USXM	ENTRY	UNFLATTEN_TRANSACTION_USER	2AF7B0BC , 00000000 , 00000000	=000353=
00060	QR	US	0301	USAD	ENTRY	ADD_USER_WITHOUT_PASSWORD_NON_TERMINAL_SIGN_ON,YES,8,CICSUSER,NO,IY2CZCCM		=000354=
00060	QR	DD	0301	DDLO	ENTRY	LOCATE	29A16660,2AC5D650,USD1,CICSUSER NO .....	=000355=
00060	QR	DD	0302	DDLO	EXIT	LOCATE/OK	2A201100 , 00000000	=000356=
00060	QR	US	0302	USAD	EXIT	ADD_USER_WITHOUT_PASSWORD/OK	0,0,0,0,00000001	=000357=
00060	QR	US	0402	USXM	EXIT	UNFLATTEN_TRANSACTION_USER/OK	00000001,2AF7B0BC , 00000000 , 00000000	=000382=
00060	QR	AS	0201	ASXM	EXIT	INIT_XM_CLIENT/OK	00000001,YES,NO	=000383=
00060	QR	US	0401	USXM	ENTRY	INIT_TRANSACTION_USER	00000001,YES	=000384=
00060	QR	XS	0401	XSMX	ENTRY	ADD_TRANSACTION_SECURITY	29956530 , 00000001	=000385=
00060	QR	XS	0402	XSMX	EXIT	ADD_TRANSACTION_SECURITY/OK		=000386=
00060	QR	US	0402	USXM	EXIT	INIT_TRANSACTION_USER/OK	2A20111F , 2A203090,0	=000387=
00060	QR	DS	0002	DSAT	ENTRY	SET_PRIORITY	1	=000388=
00060	QR	DS	0003	DSAT	EXIT	SET_PRIORITY/OK	1	=000389=
00060	QR	DP	0900	DPXM	ENTRY	INIT_XM_CLIENT		=000390=
00060	QR	DP	0901	DPXM	EXIT	INIT_XM_CLIENT/OK		=000391=
00060	QR	RM	FA01	RMUC	ENTRY	CREATE_UOW	NO,BACKWARD,0	=000392=
00060	QR	RM	0209	RMUC	EVENT	Remote_UOW_id_created	1A11C7C2C9C2D4C9E8C14BC9E8F2C3E9C3C3D41FB9DEA1FB550001	=000393=
00060	QR	RM	FA02	RMUC	EXIT	CREATE_UOW/OK		=000394=
00060	QR	FI	0B00	FI XM	ENTRY	INIT_XM_CLIENT		=000395=
00060	QR	FI	0B01	FI XM	EXIT	INIT_XM_CLIENT/OK		=000396=
00060	QR	XS	0701	XSRC	ENTRY	CHECK_CICS_RESOURCE	CHL1,TRANSATTACH,EXECUTE	=000397=
00060	QR	XS	0709	XSRC	EVENT	CHECK	CHL1 CHECK_RESOURCE_ACCESS,29956530 , 00000001,T1CVFTRN,READ,YES,2AC5	=000398=
00060	QR	XS	070A	XSRC	EVENT	CHECK-COMPLETE	CHL1 CICSUSER CHECK_RESOURCE_ACCESS,OK,0,0,0,0	=000399=
00060	QR	XS	0702	XSRC	EXIT	CHECK_CICS_RESOURCE/OK		=000400=
00060	QR	AP	0590	APXM	ENTRY	BIND_XM_CLIENT		=000401=
00060	QR	AP	0591	APXM	EXIT	BIND_XM_CLIENT/OK		=000402=
00060	QR	DP	0900	DPXM	ENTRY	BIND_XM_CLIENT		=000403=
00060	QR	DP	0901	DPXM	EXIT	BIND_XM_CLIENT/OK		=000404=
00060	QR	MN	0A01	MNXM	ENTRY	TRANSACTION_BIND	D31FB9DEA192DDBA,D31FB9DEA1D7609C,29A05CB0 , 00000200,YES	=000405=
00060	QR	MN	0A02	MNXM	EXIT	TRANSACTION_BIND/OK		=000406=
00060	QR	SM	0301	SMGF	ENTRY	FREEMAIN	299435B4 , 0000000A,29A05CA0,210,XMATODR	=000407=
00060	QR	SM	0302	SMGF	EXIT	FREEMAIN/OK		=000409=
00060	QR	AS	0200	ASXM	ENTRY	BIND_XM_CLIENT	2AF7B090 , 00000060	=000410=
00060	QR	PG	1700	PGCH	ENTRY	BIND_CHANNEL	2AF6D0B0	=000411=
00060	QR	PG	1701	PGCH	EXIT	BIND_CHANNEL/OK		=000412=
00060	QR	RM	0301	RMLN	ENTRY	ADD_LINK	ASYN,29B03A88 , 00000000 , 00000008,29B03A80 , 00000000 , 00000008,2A	=000413=
00060	QR	RM	0302	RMLN	EXIT	ADD_LINK/OK	01010001,29B03A88 , 00000000 , 00000008,29B03A80 , 00000000 , 00000000	=000414=
00060	QR	AS	0201	ASXM	EXIT	BIND_XM_CLIENT/OK	YES,CHILD1,NO,NO,	=000415=

Figure 8-21 Child CHL1 starts running

Figure 8-22 on page 140 shows the CHILD1 program running. It follows this process:

- ▶ It attempts to **GET** the 'REQUEST' container passed by the PARENT (see trace =000426=).
- ▶ However, the parent has passed a container named 'REQUESTER'. The **GET CONTAINER** command fails.
- ▶ The CHILD1 program abends with abend code BADC (see trace =000432=).



```

00059 L9000 AP 00E1 EIP ENTRY FETCH-ANY 0004,2AE008A0 .\.,09003444,0440 =000378=
00059 L9000 AP F801 EIBAM ENTRY FETCH_ANY =000379=
00059 L9000 AS 0300 ASAS ENTRY FETCH_ANY YES =000380=
00059 L9000 AS 0307 ASAS EVENT FETCH_ANY_ACCB_LIST =000381=
00059 L9000 DS 0004 DSSR ENTRY WAIT_MVS AS_ANY,2AC678A0,YES,MISC, =000408=
00059 L9000 DS 0005 DSSR EXIT WAIT_MVS/OK =001749=
00059 L9000 PG 1700 PGCH ENTRY INQUIRE_CHANNEL DFHAS00001-00060 =001750=
00059 L9000 PG 1701 PGCH EXIT INQUIRE_CHANNEL/EXCEPTION CHANNEL_NOT_FOUND =001751=
00059 L9000 PG 1700 PGCH ENTRY RENAME_CHANNEL DFHAS00001-00060,2AF6D0B0 =001752=
00059 L9000 PG 1701 PGCH EXIT RENAME_CHANNEL/OK =001753=
00059 L9000 PG 1700 PGCH ENTRY ADD_CHANNEL 2AF6D0B0,CURRENT =001754=
00059 L9000 PG 1701 PGCH EXIT ADD_CHANNEL/OK =001755=
00059 L9000 AS 0301 ASAS EXIT FETCH_ANY/OK 00000050_418000D8 , 0000060C_00000001,BADC,DFHAS00001-00060,ABENDED =001756=
00059 L9000 AP F802 EIBAM EXIT FETCH_ANY RESP=0 RESP2=0 =001757=
00059 L9000 AP 00E1 EIP EXIT FETCH-ANY OK 00F4,00000000,00003444,0240 =001758=

```

Figure 8-24 First FETCH ANY command issued by the PARENT program

Figure 8-25 shows trace =000381= formatted using the trace formatting keyword FULL. It follows this process:

- ▶ This trace is issued only when Asynchronous Services (AS) trace component 2 is on.
- ▶ The value of 00000002 in the DATA-1 section indicates that the PARENT program is waiting for the first of two children complete.
- ▶ The DATA-2 section lists the child tokens of the children that the PARENT program is about to wait for.

```

AS 0307 ASAS EVENT - FETCH_ANY_ACCB_LIST

TASK-00059 KE_NUM-004C TCB-C/L9000/009A1088 RET-2A777721 TIME-16:10:52.1570178989 INTERVAL-00.0000006782 =000381=
1-0000 00000002 *.....*
2-0000 00000050 418000D8 0000060C 00000001 00000050 41800048 0000061C 00000002 *...&...Q.....&.....*

```

Figure 8-25 FETCH ANY WAIT event trace

## 8.5 Sample application trace flow using FETCH CHILD commands and the NOSUSPEND and TIMEOUT options

This section shows selected trace output of a sample asynchronous API application that we ran where the parent used the EXEC CICS FETCH CHILD command. The trace was directed to a CICS auxiliary trace data set and formatted using a CICS supplied auxiliary trace formatter. All the trace entries shown were formatted with the ABBREV keyword unless otherwise stated.

### 8.5.1 The environment

We defined a parent transaction (PRNT) that used an assembler program called PARENT, as shown in Figure 8-26 on page 142. The parent used the EXEC CICS RUN TRANSID command to create a single child transaction named CHLD.

The PARENT program then issued the following commands:

- ▶ EXEC CICS FETCH CHILD NOSUSPEND, completes with NOTFINISHED
- ▶ EXEC CICS FETCH CHILD TIMEOUT(1-second), completes with NOTFINISHED
- ▶ EXEC CICS FETCH CHILD, blocks until the CHLD task completes

```

DFHEISTG DSECT
CHLDTOK DS CL16 CHILD TOKEN FOR CHLD
FETCHRESPCHAN DS CL16 RESPONSE CHANNEL ON FETCH COMMAND
ABCODE DS CL4 ABEND CODE ON FETCH COMMAND
COMPSTAT DS CL4 COMPLETION STATUS ON FETCH COMMAND
PARENT CSECT
PARENT AMODE 31
PARENT RMODE ANY
RUNCHLD DS 0H
 EXEC CICS PUT CONTAINER('REQUEST') FROM(REQ) *
 CHANNEL('CHLD-CHANNEL')
 EXEC CICS RUN TRANSID('CHLD') CHILD(CHLDTOK) *
 CHANNEL('CHLD-CHANNEL')
FETCHNOS DS 0H
 EXEC CICS FETCH CHILD(CHLDTOK) CHANNEL(FETCHRESPCHAN) *
 COMPSTATUS(COMPSTAT) ABCODE(ABCODE) NOSUSPEND NOHANDLE
FETCHTIM DS 0H
 EXEC CICS FETCH CHILD(CHLDTOK) CHANNEL(FETCHRESPCHAN) *
 COMPSTATUS(COMPSTAT) ABCODE(ABCODE) TIMEOUT(TIMEOUT) *
 NOHANDLE
FETCH DS 0H
 EXEC CICS FETCH CHILD(CHLDTOK) CHANNEL(FETCHRESPCHAN) *
 COMPSTATUS(COMPSTAT) ABCODE(ABCODE) NOHANDLE
 EXEC CICS SEND TEXT FROM(TERMMSG) FREEKB ERASE
 EXEC CICS RETURN
TIMEOUT DC AL4(1000) 1000 milliseconds timeout
TERMMSG DC CL30'PARENT TRANSACTION COMPLETE'
REQ DC CL30'PARENT REQUEST FOR CHILD'
 END PARENT

```

Figure 8-26 The PARENT program using EXEC CICS FETCH CHILD commands

The CHLD task ran the CHILD assembler program. The default transaction and program attributes were used. The code for the CHILD program is shown in Figure 8-27. The CHILD program issues a 2-second delay before completing.

```

DFHEISTG DSECT
FLEN DS F
CONTAREA DS CL30
CHILD CSECT
CHILD AMODE 31
CHILD RMODE ANY
 EXEC CICS DELAY INTERVAL(2)
 MVC FLEN,=F'30'
 EXEC CICS GET CONTAINER('REQUEST') INTO(CONTAREA) *
 FLLENGTH(FLEN) NOHANDLE
 CLC EIBRESP,DFHRESP(NORMAL)
 BNE ABEND
 CLC CONTAREA,REQ
 BNE ABEND
 EXEC CICS PUT CONTAINER('RESPONSE') FROM(RES)
RETURN DS 0H
 EXEC CICS RETURN
ABEND DS 0H
 EXEC CICS ABEND ABCODE('BAD')
REQ DC CL30'PARENT REQUEST FOR CHILD'
RESP DC CL30'NORMAL RESPONSE FROM CHLD'
 END CHILD

```

Figure 8-27 The CHILD program with the 2-second delay

## 8.5.2 Trace of FETCH CHILD NO SUSPEND

Figure 8-28 shows a trace of the **EXEC CICS FETCH CHILD NOSUSPEND** command issued by the PARENT program. This trace fails with a NOTFINISHED response because the CHLD task has not completed.

00071	L9000	AP	00E1	EIP	ENTRY	FETCH-CHILD		0004,2AE008A0 .\.,09003442 ....,0440	=000360=
00071	L9000	AP	F801	EIBAM	ENTRY	FETCH_CHILD			=000361=
00071	L9000	AS	0300	ASAS	ENTRY	FETCH_CHILD	00000050_41800048 , 0000072C_00000001,NO		=000362=
00071	L9000	AS	0301	ASAS	EXIT	FETCH_CHILD/EXCEPTION	NOTFINISHED,,,		=000363=
00071	L9000	AP	F802	EIBAM	EXIT	FETCH_CHILD	RESP=113 RESP2=52		=000364=
00071	L9000	AP	00E1	EIP	EXIT	FETCH-CHILD	NOTFIN	00F4,00000034 ....,00713442 ....,0240	=000365=

Figure 8-28 EXEC CICS FETCH CHILD NOSUSPEND fails with a NOTFINISHED response

## 8.5.3 Trace of FETCH CHILD TIMEOUT

Figure 8-29 shows a trace of the **EXEC CICS FETCH CHILD TIMEOUT** command issued by the PARENT program. It follows this process:

- ▶ The trace eventually fails with a NOTFINISHED response because the CHLD task does not complete within the 1-second timeout period.
- ▶ Trace =000369= shows the parent waiting for the child.
- ▶ The wait type is AS\_CHILD. This wait type contrasts with the wait type of AS\_ANY, which is used on **EXEC CICS FETCH ANY** commands.

00071	L9000	AP	00E1	EIP	ENTRY	FETCH-CHILD		0004,2AE008A0 .\.,09003442 ....,0440	=000366=
00071	L9000	AP	F801	EIBAM	ENTRY	FETCH_CHILD			=000367=
00071	L9000	AS	0300	ASAS	ENTRY	FETCH_CHILD	00000050_41800048 , 0000072C_00000001,YES,3E8		=000368=
00071	L9000	DS	0004	DSSR	ENTRY	WAIT_MVS	AS_CHILD,3E8,2AF9A040,YES,MILLI_SECOND,MISC,00000072		=000369=
00071	L9000	DS	0005	DSSR	EXIT	WAIT_MVS/PURGED	TIMED_OUT		*=000506=
00071	L9000	AS	0301	ASAS	EXIT	FETCH_CHILD/EXCEPTION	TIMED_OUT,,,		=000507=
00071	L9000	AP	F802	EIBAM	EXIT	FETCH_CHILD	RESP=113 RESP2=53		=000508=
00071	L9000	AP	00E1	EIP	EXIT	FETCH-CHILD	NOTFIN	00F4,00000035 ....,00713442 ....,0240	=000509=

Figure 8-29 EXEC CICS FETCH CHILD TIMEOUT() fails with NOTFINISHED response

## 8.5.4 Trace of FETCH CHILD

Figure 8-30 shows a trace of the last **EXEC CICS FETCH CHILD** command issued by the PARENT program. This trace waits until the CHLD task completes (see trace =000513=).

00071	L9000	AP	00E1	EIP	ENTRY	FETCH-CHILD		0004,2AE008A0 .\.,09003442 ....,0440	=000510=
00071	L9000	AP	F801	EIBAM	ENTRY	FETCH_CHILD			=000511=
00071	L9000	AS	0300	ASAS	ENTRY	FETCH_CHILD	00000050_41800048 , 0000072C_00000001,YES		=000512=
00071	L9000	DS	0004	DSSR	ENTRY	WAIT_MVS	AS_CHILD,2AF9A040,YES,MISC,00000072		=000513=
00071	L9000	DS	0005	DSSR	EXIT	WAIT_MVS/OK			=000606=
00071	L9000	PG	1700	PGCH	ENTRY	INQUIRE_CHANNEL	DFHAS00001-00072		=000607=
00071	L9000	PG	1701	PGCH	EXIT	INQUIRE_CHANNEL/EXCEPTION	CHANNEL_NOT_FOUND		=000608=
00071	L9000	PG	1700	PGCH	ENTRY	RENAME_CHANNEL	DFHAS00001-00072,2AF9A070		=000609=
00071	L9000	PG	1701	PGCH	EXIT	RENAME_CHANNEL/OK			=000610=
00071	L9000	PG	1700	PGCH	ENTRY	ADD_CHANNEL	2AF9A070,CURRENT		=000611=
00071	L9000	PG	1701	PGCH	EXIT	ADD_CHANNEL/OK			=000612=
00071	L9000	AS	0301	ASAS	EXIT	FETCH_CHILD/OK	,DFHAS00001-00072,NORMAL		=000613=
00071	L9000	AP	F802	EIBAM	EXIT	FETCH_CHILD	RESP=0 RESP2=0		=000614=
00071	L9000	AP	00E1	EIP	EXIT	FETCH-CHILD	OK	00F4,00000000 ....,00003442 ....,0240	=000615=

Figure 8-30 EXEC CICS FETCH CHILD blocks until CHLD completes



- ▶ The CHILD program issues a 2-second delay.
- ▶ The CHLD task then terminates (see trace =000571=).

Figure 8-31 The CHILD program delays for 2-seconds before completing

This section shows selected trace output of a sample asynchronous API application that we ran where the parent used **EXEC CICS FREE CHILD** commands to free its two children. The trace was directed to a CICS auxiliary trace data set and formatted using a CICS supplied auxiliary trace formatter. All the trace entries shown were formatted with the **ABBREV** keyword unless otherwise stated.

## 8.6.1 The environment

We defined a parent transaction (PRNT) that used an assembler program called PARENT as shown in Figure 8-32. It followed this process:

- ▶ The PRNT transaction used **EXEC CICS RUN TRANSID** commands to create two child tasks, CHL1 and CHL2, and passed an empty channel to each one.
- ▶ The PRNT transaction then delayed for 1-second.
- ▶ After this delay, the CHL1 task had completed, but the CHL2 task was still active, as shown in Figure 8-33.
- ▶ The PRNT transaction then freed the CHL1 and CHL2 tasks, as shown in Figure 8-34.

```
DFHEISTG DSECT
CHL1TOK DS CL16
CHL2TOK DS CL16
PARENT CSECT
PARENT AMODE 31
PARENT RMODE ANY
EXEC CICS RUN TRANSID('CHL1') CHANNEL('CHAN') CHILD(CHL1TOK)
EXEC CICS RUN TRANSID('CHL2') CHANNEL('CHAN') CHILD(CHL2TOK)
EXEC CICS DELAY INTERVAL(1)
EXEC CICS FREE CHILD(CHL1TOK)
EXEC CICS FREE CHILD(CHL2TOK)
EXEC CICS RETURN
END PARENT
```

Figure 8-32 The PARENT program frees the CHL1 and CHL2 tasks

```
CHILD1 CSECT
CHILD1 AMODE 31
CHILD1 RMODE ANY
EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESF)
EXEC CICS RETURN
RESF DC CL30 'NORMAL RESPONSE FROM CHL1'
END CHILD1
```

Figure 8-33 The CHLD1 program completes immediately

```
CHILD2 CSECT
CHILD2 AMODE 31
CHILD2 RMODE ANY
EXEC CICS DELAY INTERVAL(2)
EXEC CICS PUT CONTAINER('RESPONSE') FROM(RESF)
EXEC CICS RETURN
RESF DC CL30 'NORMAL RESPONSE FROM CHL2'
END CHILD2
```

Figure 8-34 The CHLD2 program delays for two seconds

## 8.6.2 Trace of free child tasks

Figure 8-35 on page 146 shows the PRNT transaction freeing the CHL1 task (trace =000854=) and the CHL2 task (trace =000872=). Neither of these children had been fetched. The CHL1 task had already completed, so the **FREE CHILD** command cleaned up the child state, including the child channel. The CHL2 task had not completed, and child cleanup was deferred until the CHL2 task completed. Trace =000875= shows that the CHL2 task had not completed.

00049	L9000	AP	00E1	EIP	ENTRY	FREE-CHILD		0004,2AE008A0 .\.,09003446 ....,0440	=000854=
00049	L9000	AP	F801	EIBAM	ENTRY	FREE CHILD			=000855=
00049	L9000	AS	0300	ASAS	ENTRY	FREE CHILD	00000050_41800048 , 0000050C_00000001		=000856=
00049	L9000	PG	1700	PGCH	ENTRY	DELETE CHANNEL	2AF7E030		=000857=
00049	L9000	PG	1800	PGCP	ENTRY	DELETE CONTAINER POOL	2AF7F030		=000858=
00049	L9000	PG	1900	PGCR	ENTRY	DELETE CONTAINER	2AF92030,SYSTEM		=000859=
00049	L9000	SM	4201	S2GF	ENTRY	FREEMAIN	00000050_40904AE8 , 00000000_00000074,00000050_41000000,1000,CSDB,YES		=000860=
00049	L9000	SM	4202	S2GF	EXIT	FREEMAIN/OK			=000861=
00049	L9000	PG	1901	PGCR	EXIT	DELETE CONTAINER/OK			=000862=
00049	L9000	SM	0301	SMGF	ENTRY	FREEMAIN	48E404E8 , 00000070,2AF7F030,30,CPCB,YES		=000863=
00049	L9000	SM	0302	SMGF	EXIT	FREEMAIN/OK			=000864=
00049	L9000	PG	1801	PGCP	EXIT	DELETE CONTAINER POOL/OK			=000865=
00049	L9000	SM	0301	SMGF	ENTRY	FREEMAIN	48E4041C , 0000006F,2AF7E030,40,CHCB,YES		=000866=
00049	L9000	SM	0302	SMGF	EXIT	FREEMAIN/OK			=000867=
00049	L9000	PG	1701	PGCH	EXIT	DELETE CHANNEL/OK			=000868=
00049	L9000	AS	0301	ASAS	EXIT	FREE CHILD/OK			=000869=
00049	L9000	AP	F802	EIBAM	EXIT	FREE CHILD	RESP=0 RESP2=0		=000870=
00049	L9000	AP	00E1	EIP	EXIT	FREE-CHILD	OK	00F4,00000000 ....,00003446 ....,0240	=000871=
00049	L9000	AP	00E1	EIP	ENTRY	FREE-CHILD		0004,2AE008A0 .\.,09003446 ....,0440	=000872=
00049	L9000	AP	F801	EIBAM	ENTRY	FREE CHILD			=000873=
00049	L9000	AS	0300	ASAS	ENTRY	FREE CHILD	00000050_418000D8 , 0000051C_00000002		=000874=
00049	L9000	AS	0309	ASAS	EVENT	CHILD NOT FINISHED WHEN FREED			=000875=
00049	L9000	AS	0301	ASAS	EXIT	FREE CHILD/OK			=000876=
00049	L9000	AP	F802	EIBAM	EXIT	FREE CHILD	RESP=0 RESP2=0		=000877=
00049	L9000	AP	00E1	EIP	EXIT	FREE-CHILD	OK	00F4,00000000 ....,00003446 ....,0240	=000878=

Figure 8-35 Trace of the PRNT transaction freeing the CHL1 and CHL2 tasks

## 8.7 Transaction dumps and the asynchronous API

This section shows two examples of transaction dumps taken by an asynchronous parent task and an asynchronous child task. All the dumps were taken because of transaction abends.

**Tip:** Set the SIT parameter to TRTRANITY=ALL to assist with debugging by using transaction dumps and the asynchronous API. This option ensures that all tasks show in the trace table, which is included in a transaction dump, rather than just the dumping task.

### 8.7.1 Asynchronous parent task transaction dump extract

The following example extracts from a transaction dump taken by an asynchronous parent task PRNT that abended with abend code PRNT. The same information appears if the parent task took a dump taken using the **EXEC CICS DUMP TRANSACTION DUMPCODE() COMPLETE** command.

In this example, the parent transaction ID was PRNT, and it was task number 00061. The parent created five children, transaction IDs CHL1, CHL2, CHL3, CHL4, and CHL5.

At the time of the transaction dump, the five child tasks were in the following states:

- ▶ CHL1: Task number 00062 was abended.
- ▶ CHL2: Task number 00063 was still active.
- ▶ CHL3: Task number 00064 completed normally.
- ▶ CHL4: Task number 00065 was not started because it was in a TRANCLASS that had reached its MAXACTIVE threshold.
- ▶ CHL5: Task number 00066 was freed by an **EXEC CICS FREE CHILD** command issued by the parent.

The dump was formatted using the CICS supplied transaction dump utility program.



Figure 8-36 shows PAGE-1 of the formatted transaction dump taken by the parent. It shows basic information about the PRNT task such as task number, abend code and terminal ID.

```

IY2C2CCM --- CICS TRANSACTION DUMP --- CODE=PRNT TRAN=PRNT ID=1/0006 DATE=17/09/19 TIME=10:06:26 PAGE 1
SYMPTOMS= AB/UPRNT PIDS/5655Y0400 FLDS/DFHABAB RIDS/PARENT
CICS LEVEL = 0720
REGISTERS AT LAST EXEC COMMAND
REGS 0-7 00000000 2AE00908 00041800 AC001028 2AC6EC40 2AC6F0BC 2A32650A 2AC6F298
REGS 8-15 48E2C680 29A8A810 2AE00008 2AE00100 009A9000 2AE008A0 AC001740 00000000

Transaction environment for transaction_number(0000061)
transaction_id(PRNT) orig_transaction_id(PRNT)
initial_program(PARENT) current_program(PARENT)
facility_type(TERMINAL) facility_name(TC17) Start_code(T0)
netname(IYFHTC17) profile_name(DFHCICST)
userid(HORN) cmdsec(NO) ressec(YES)
spurge(YES) dtimeout(00000000) tpurge(YES)
taskdatakey(USER) taskdataloc(ANY)
twasize(000000) twaaddr()
remote(NO) dynamic(NO)
priority(001) Tclass(YES,JULS) runaway_limit(0005000)
indoubt_wait(YES) indoubt_wait_mins(000000)
indoubt_action(BACKOUT) cics_uow_id(D329775215351904) confdata(NO)
system_transaction(NO) restart_count(000000) restart(NO)

```

Figure 8-36 PAGE-1 of formatted transaction dump taken by the PRNT task

Figure 8-37 shows the formatted asynchronous API state from the same transaction dump.

```

ASYNCHRONOUS CHILD SUMMARY FOR THE CURRENT TRANSACTION
TOTAL NUMBER OF CHILDREN CREATED : 00000005
TOTAL NUMBER OF CHILDREN FREED : 00000001
ASYNCHRONOUS CHILD LIST
Child_Trannum(00062) Child_Tranid(CHL1) Child_State(ABENDED) Child_Token(00000050_418001F8 , 0000062C_00000001)
Child_Trannum(00063) Child_Tranid(CHL2) Child_State(ACTIVE) Child_Token(00000050_41800288 , 0000063C_00000002)
Child_Trannum(00064) Child_Tranid(CHL3) Child_State(COMPLETED) Child_Token(00000050_41800168 , 0000064C_00000003)
Child_Trannum(00065) Child_Tranid(CHL4) Child_State(NOTSTARTED) Child_Token(00000050_418000D8 , 0000065C_00000004)

```

Figure 8-37 Asynchronous API state from formatted transaction dump taken by the PRNT task

The formatted transaction dump includes the following information:

- The summary states how many child tasks the parent created and how many are freed.
- There are no details for the CHL5 task in the ASYNCHRONOUS CHILD LIST because it was freed by the parent before the dump was taken. All other children are listed.
- All the possible transaction states appear. These are ACTIVE, ABENDED, COMPLETED (without ABEND), and NOTSTARTED.

## 8.7.2 Asynchronous child task transaction dump extract

The following example shows extracts from a transaction dump taken by an asynchronous child task CHL1 that abended with abend code BADC. The same information appears if the parent task took a dump taken using the `EXEC CICS DUMP TRANSACTION DUMPCODE() COMPLETE` command.

In this example, the child transaction ID was CHL1, and it was task number 00062. Its parent (transaction ID PRNT) was task number 00061.

The dump was formatted using the CICS supplied transaction dump utility program.

Figure 8-38 shows PAGE-1 of the formatted transaction dump taken by the child. It shows the task number and the abend code of the child. The information about the parent task displays in the last 3-lines of Figure 8-38. Possible parent states are ACTIVE, ABENDED, and COMPLETED.

IY2CZCCM	---	CICS TRANSACTION DUMP	---	CODE=BADC	TRAN=CHL1	ID=1/0004	DATE=17/09/19	TIME=10:06:22	PAGE	1
SYMPTOMS= AB/UBADC PIDS/5655Y0400 FLDS/DFHABAB RIDS/CHLD1										
CICS LEVEL = 0720										
REGISTERS AT LAST EXEC COMMAND										
REGS 0-7	2C200A24	2C200908	00041800	AC002338		2AC56C40	2AC570BC	2A32650A	2AC57298	
REGS 8-15	7F615290	2AC53000	2C200008	2C200100		009A9000	2C2008A0	AC002512	00000000	
Transaction environment for transaction_number(0000062)										
transaction_id(CHL1)		orig_transaction_id(CHL1)								
initial_program(CHILD1 )		current_program(CHILD1 )								
facility_type(NONE)		facility_name( )		Start_code(U )						
userid(HORN )		cmdsec(NO)		ressec(NO)						
spurge(YES)		dtimeout(0000000)		tpurge(YES)						
taskdatakey(USER)		taskdataloc(ANY)								
twasize(00000)		twaaddr( )								
remote(NO)		dynamic(NO)								
priority(001)		Tclass(NO)		runaway_limit(0005000)						
indoubt_wait(YES)		indoubt_wait_mins(000000)								
indoubt_action(BACKOUT)		cics_uow_id(D32977521561BF04)		confdata(NO)						
system_transaction(NO)		restart_count(00000)		restart(NO)						
This transaction is running as an ASYNCHRONOUS child task										
Child_Token(00000050_418001F8 , 00000062C_00000001)										
Parent_Trannum(00061)		Parent_Tranid(PRNT)		Parent_State(ACTIVE )						

Figure 8-38 PAGE-1 of formatted transaction dump taken by the CHL1 child task

## 8.8 System dumps and the asynchronous API

This section shows an example of a formatted system dump. It contains a single asynchronous parent task and five asynchronous children.

### 8.8.1 Asynchronous parent system dump extract

This example uses the same scenario described in 8.7.1, “Asynchronous parent task transaction dump extract” on page 146. There is a single parent transaction (PRNT) that created five children. The PRNT parent transaction took a system dump at the same point it took a transaction dump.

In this example, the parent transaction ID was PRNT, and it was task number 00061. The parent had created five children transactions IDs, CHL1, CHL2, CHL3, CHL4, and CHL5.

At the time of the system dump, the five child tasks were in the following states:

- ▶ CHL1: Task number 00062 was abended.
- ▶ CHL2: Task number 00063 was still active.
- ▶ CHL3: Task number 00064 completed normally.
- ▶ CHL4: Task number 00065 was not started because it was in a TRANCLASS, which had reached its MAXACTIVE threshold.
- ▶ CHL5: Task number 00066 was freed by an **EXEC CICS FREE CHILD** command issued by the parent. This command causes CHL5 to become an orphaned child.

The dump was formatted using the CICS supplied system dump **verbexit** program, which is invoked from the interactive problem control system (IPCS).

Figure 8-39 shows the PARENT TASKS SUMMARY section of the system dump formatted with the AS=1 keyword of the system dump formatter. This summary shows the single parent task (PRNT) and all of its children, apart from the CHL5 task, which was freed and, therefore, is an orphan.

===AS: PARENT TASKS SUMMARY													
Key for ACCB summary table:													
Child status : ACT=Active, COM=Completed, AB=Abended, SECERR=XTRAN security error, NST=Not started													
Parent status : ACT=Active, COM=Completed, AB=Abended, FC=Freed child													
Child Fetched : Y=Child fetched by parent, N=Child not fetched by parent													
Channel used : Y=Child started with channel, N=Child started without channel													
==AS: ACTXN SUMMARY FOR PARENT TRAN NUM : 00061													
TOTAL NUMBER OF CHILD TASKS CREATED : 5													
TOTAL NUMBER OF CHILD TASKS FREED : 1													
=AS: ACCB SUMMARY													
ACCB Address	Child Tran num	Child Tran id	Child Status	Parent Tran num	Parent Tran id	Child num	Parent Status	Child Fetched	Channel used	Fetched channel	ACCB Date	Creation Time	
00000050_418001F8	00062	CHL1	AB	00061	PRNT	1	ACT	Y	Y	DFHAS00001-00062	19/09/17	09:06:23	
00000050_41800288	00063	CHL2	ACT	00061	PRNT	2	ACT	N	Y		19/09/17	09:06:23	
00000050_41800168	00064	CHL3	COM	00061	PRNT	3	ACT	Y	Y	DFHAS00003-00064	19/09/17	09:06:23	
00000050_418000D8	00065	CHL4	NST	00061	PRNT	4	ACT	N	Y		19/09/17	09:06:23	

Figure 8-39 PARENT TASKS SUMMARY of system dump formatted with the AS=1 keyword

Figure 8-40 shows the ORPHANED CHILD TASKS SUMMARY section of the system dump, which is also formatted with the AS=1 keyword.

===AS: ORPHANED CHILD TASKS SUMMARY													
Key for ACCB summary table:													
Child status : ACT=Active, COM=Completed, AB=Abended, SECERR=XTRAN security error, NST=Not started													
Parent status : ACT=Active, COM=Completed, AB=Abended, FC=Freed child													
Child Fetched : Y=Child fetched by parent, N=Child not fetched by parent													
Channel used : Y=Child started with channel, N=Child started without channel													
ACCB Address	Child Tran num	Child Tran id	Child Status	Parent Tran num	Parent Tran id	Child num	Parent Status	Child Fetched	Channel used	Fetched channel	ACCB Date	Creation Time	
00000050_41800048	00066	CHL5	ACT	00061	PRNT	5	FC	N	Y		19/09/17	09:06:23	

Figure 8-40 ORPHANED CHILD TASKS SUMMARY of system dump formatted with the AS=1 keyword





## Performance and management for asynchronous API applications

The asynchronous API brings the benefit of reduced response time to applications. The CICS asynchronous API solution also gives you an easier environment to manage compared to other asynchronous solutions.

This chapter looks at the performance aspects of an asynchronous API application, apart from the general CICS performance, and what impact to consider to support this kind of application. It also addresses how to manage this kind of application. This chapter includes the following topics:

- ▶ Special aspects for asynchronous API applications
- ▶ Using MXT
- ▶ Duration of parent tasks in the system
- ▶ Policing parent tasks with CICS policy
- ▶ Policing parent tasks with CICS policy
- ▶ Threadsafe considerations
- ▶ Asynchronous services statistics
- ▶ Asynchronous services monitoring

## 9.1 Special aspects for asynchronous API applications

An asynchronous API application starts one or more child tasks. When fetching a child response, the parent task remains in the system while the child task is running. This process is different from the application that issues an **EXEC CICS START** command, which starts a task but doesn't fetch the response from the started task. The task issuing the **EXEC CICS START** command can complete while the started task is still running.

An asynchronous API application differs also from an application that uses the **EXEC CICS LINK** command. A task that issues the **EXEC CICS LINK** command doesn't start another task. Instead, it remains in the system until the linked program is finished.

Therefore, an asynchronous API application provides the following additional aspects to consider when thinking about the performance impact to the system:

- ▶ How many child tasks can be spawned by a parent task?
- ▶ How long can a parent task likely remain in the system?

## 9.2 Managing the number of tasks in the system

This chapter addresses the following:

- ▶ Using MXT
- ▶ Using TRANCLASS to manage parent transactions
- ▶ Using TRANCLASS to manage child transactions

### 9.2.1 Using MXT

Tasks in a CICS system occupy resources, such as task slots and storage. If the system has too many tasks, they will compete for these resources. The maximum task specification (MXT) in a system controls the number of user tasks that are eligible for dispatch.

If setting the MXT value too high, tasks might not be able to timely obtain the system resources that they needed, such as CPU and virtual storage. Contention can occur for resources, such as files, buffer, IBM DB2® threads, and so on. Short-on-storage (SOS) condition can arise when too many tasks are competing for virtual storage. Tasks can be queued or abend when in this situation. Then, the user experiences unresponsiveness of the system.

If setting the MXT value too low, tasks might be delayed due to excessive queuing and waiting to be dispatched. As a result, the system resource is under utilized, and the user experiences a slower response from the system.

Therefore, setting the MXT appropriately is important for the system and the user. As a starting point, for a system that already has a number of active tasks (say,  $m$ ), to cater the number of new tasks brought in by asynchronous application, you need to determine the number of parent tasks (say,  $p$ ) and how many child tasks that can run in parallel to a parent task (say,  $c$ ). After knowing these values, the MXT of the system can be set accordingly using a formula, such as:

$$p * (1+c) + m$$

This value of MXT can be a good starting point for the system. Depending on what child tasks do, for example, if these child tasks also become parent tasks, you might need to factor that in and change this value.

You also need to configure virtual storage, and CPU accordingly. You can monitor how these resources are used and whether any SOS condition occurs. Then, increase storage when necessary so that the SOS condition doesn't occur when the MXT number of tasks is reached.

## 9.2.2 Using TRANCLASS to manage parent transactions

It is advisable to limit the number of parent tasks so that the number of tasks, including the potential child tasks, in the system doesn't exceed the limit that MXT defines. Limiting the number of parent tasks effectively limits the number of child tasks in the system too.

The transaction class resource (TRANCLASS) has an attribute of MAXACTIVE, which is used to constrain the number of active tasks for transactions that belong to the TRANCLASS. You can use this attribute to limit the number of parent tasks.

For example, CICS has a MXT value of 400 and already runs 200 tasks at peak time. When you deploy a new asynchronous application to this system, which can start to four children to run concurrently in the system, the MAXACTIVE attribute for the TRANCLASS of the parent transaction is calculated as:

$$(400 - 200) / (1 \text{ parent task} + 4 \text{ child tasks}) = 40$$

By defining the MAXACTIVE to a value that allows the system to cater for the maximum number of concurrent children and parents, the system should be able to serve workload smoothly.

You can use ASRUNCT monitoring field in the DFHTASK group of the CICS monitoring performance class to get an idea of how many child tasks that parent transaction started. For more information, refer to [IBM Knowledge Center](#).

**Tip:** Do not assign the child transaction to the same TRANCLASS transaction class as its parent transaction. If you do that, the child transaction might not be able to start because the MAXACTIVE limit is reached. Delay of a child transaction can cause its parent to wait for a response and to occupy a slot in the MXT and MAXACTIVE limit.

## 9.2.3 Using TRANCLASS to manage child transactions

You might wonder what the control for the child transaction should be. The number of child transactions is decided by the application logic of its parent transaction. For the parent transaction to finish as soon as possible, do not assign the child transaction to a TRANCLASS transaction class that might limit the number of child transactions that run and, in turn, delay the parent transaction.

The transaction priority of the child transaction needs to be high enough so that it is not delayed for being dispatched.

## 9.3 Duration of parent tasks in the system

The parent application needs a well-designed logic flow so that the parent task doesn't need to wait indefinitely for the child task to finish when it comes to the time of fetching the child

response. When the application is well-designed and runs in a well-tuned system, tasks in the system finish without undue delay so that the response time and the transaction rate are not compromised. When the system is not well-tuned or the application is not well-designed, a parent task can be delayed.

This section discusses possible delays and how to discover the cause of the delays by looking at CICS monitoring and statistics.

### 9.3.1 Parent tasks waiting upon child tasks

Asynchronous parent tasks might need to get responses from their child tasks. To avoid delays in the parent tasks because of the slow responses from child tasks, the child tasks should ideally start as soon as possible.

If a parent task is suspended because it is fetching a child response, the **CEMT INQ TASK** command shows the parent task's status as Runstatus(Suspended) and Htype with the AS\_CHILD or AS\_ANY resource type depending on whether the parent task issued a **FETCH CHILD** or **FETCH ANY** command.

You can also look at ASFTCHWT field in the DFHTASK group of the CICS monitoring performance class of the parent tasks to determine whether a parent task was delayed because of waiting for child task to finish. (For more information, refer to [IBM Knowledge Center](#) and 9.7, "Asynchronous services monitoring" on page 156.)

It might be normal if the ASFTCHWT field is non-zero. However the field becomes large, which can indicate that an unduly large delay occurred, check why child tasks were delayed for so long and investigate how the delay can be reduced. For ways to determine the child tasks and their response time from CICS monitoring performance class SMF 110 records, refer to 10.4, "Using CICS Performance Analyzer to understand task relationship" on page 171. You might need to examine the application code to see if the logic flow can be improved to reduce the delay.

### 9.3.2 MAXTASK condition causing parent tasks to suspend

Another possible delay of the parent task can be caused by the system reaching the MXT limit while the parent task tries to start a child task by issuing RUN TRANSID command. In this case, CICS automatically regulates the load by suspending the parent task until the system overload situation is relieved. If a parent task becomes suspended because of system overload, the **CEMT INQ TASK** command shows the parent task's status as Runstatus(Suspended) and Htype(ASPARENT).

You can also observe the delay of parent task caused by the system reaching the MXT limit by looking at the ASRNATWT field in DFHTASK group of the CICS monitoring performance class of the parent tasks.

From a region level, you can observe whether the system becomes overloaded, causing parent tasks to be delayed in the asynchronous services global statistics report that is produced by the DFHSTUP or DFHOSTAT programs. The following fields in the report provide the detailed figures:

- ▶ The number of times the **RUN TRANSID** command is delayed (ASG\_RUN\_DELAY\_COUNT). One instance of **RUN TRANSID** can be delayed for multiple times when the system load fluctuates at the border of MXT.
- ▶ Current parents delayed (ASG\_PARENTS\_DELAYED\_CUR)
- ▶ Peak parents delayed (ASG\_PARENTS\_DELAYED\_PEAK)



If this kind of delay happened, investigate what has caused the MXT condition and take actions such as try to increase the MXT value of the system or reduce the number of parent tasks so the overload is removed.

## 9.4 Policing parent tasks with CICS policy

Even in a well-tuned system, situations can arise where you can encounter unexpected behavior from parent tasks that start more child tasks than expected. The system can become overloaded because of excessive number of child tasks.

You can use the CICS policy to monitor and detect this kind of situation. CICS supports a task policy with a Async requests rule type, where you can define a trigger threshold of the number of **EXEC CICS RUN TRANSID** commands for a user task and either emit a message, trigger an event, or abend the task.

Figure 9-1 shows the CICS Explorer interface for defining such a rule, which detects parent tasks issuing over 10 **EXEC CICS RUN TRANSID** commands and abends them with an AMPB abend code.

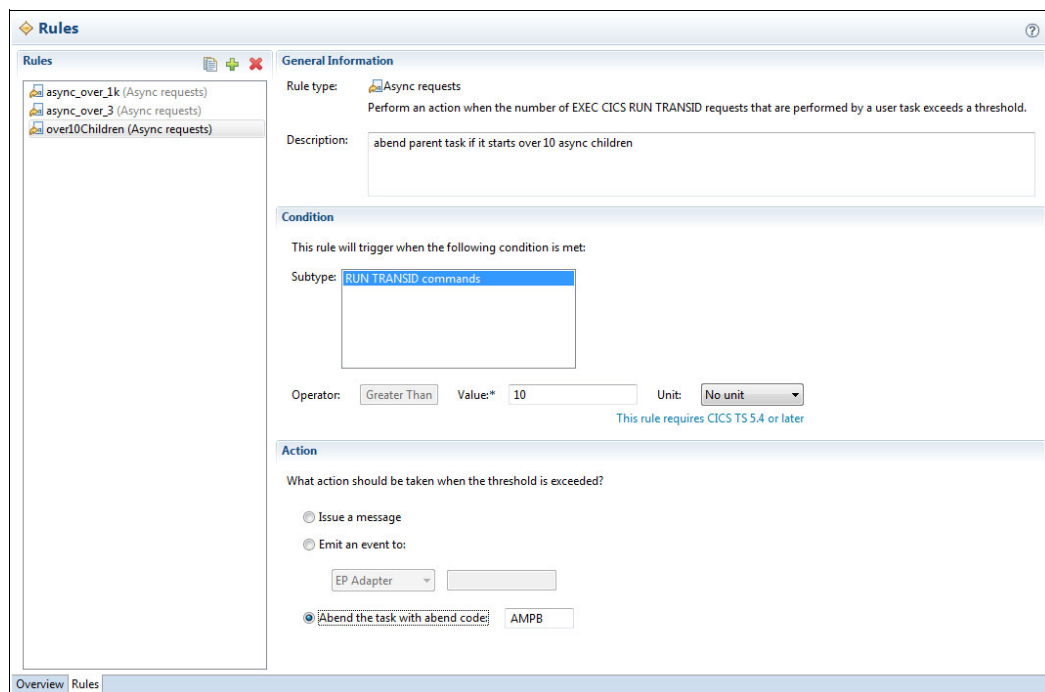


Figure 9-1 Async request policy rule

After defined the policy in a CICS bundle, you can install the bundle to a CICS system which will then police the parent tasks. For details about CICS policy, refer to [IBM Knowledge Center](#).

## 9.5 Threadsafe considerations

The asynchronous API commands are threadsafe. With threadsafe commands, you can use it with other threadsafe commands and CICS does not need to switch between TCBs, therefore saving CPU time and response time. When the command is not threadsafe, the task is forced

to run on the quasi-reentrant (QR) task control block (TCB), therefore competing with other tasks running on the QR TCB.

You can find more information about threadsafe in [IBM Knowledge Center](#).

*IBM CICS Performance Series: CICS TS for z/OS V5 Performance Report, SG24-8298* made a comparison between the **EXEC CICS START** command and the **EXEC CICS RUN TRANSID** command. The **EXEC CICS RUN TRANSID** performs better than the non-threadsafe **EXEC CICS START** command.

## 9.6 Asynchronous services statistics

To monitor a CICS region that has an asynchronous API application, you can turn on statistics. You can view asynchronous services domain global statistics online with the CICS-supplied sample program DFHOSTAT or offline by using the statistics utility program DFHSTUP.

Example 9-1 shows an example of asynchronous services domain global statistics. With 744 times of **RUN TRANSID** commands being delayed, it is evident that some parent tasks in this region were delayed due to the system being at the MXT. The delay can result in longer response time for the user that started these parent tasks. You might need to consider increasing the system's MXT value or reducing the rate of parent transactions being started by directing some of the requests to another CICS system.

*Example 9-1 Asynchronous services domain global statistics*

---

### Asynchronous services

RUN commands. . . . .	4,026
FETCH commands. . . . .	11
FREE commands . . . . .	0
Current active children . . . . .	0
Peak active children. . . . .	7
Times RUN commands being delayed. . .	744
Current parents being delayed . . .	0
Peak parents being delayed. . . . .	1

---

## 9.7 Asynchronous services monitoring

To monitor tasks that issue asynchronous API commands, monitoring fields are added to DFHTASK group. Table 9-1 lists these fields.

*Table 9-1 Performance data in DFHTASK group*

Field name	Field ID	Description
ASTOTCT	470	The total number of <b>EXEC CICS</b> asynchronous API commands that have been issued by the user task. Includes the <b>RUN TRANSID</b> , <b>FETCH CHILD</b> , <b>FETCH ANY</b> , and <b>FREE CHILD</b> commands.
ASRUNCT	471	The number of <b>EXEC CICS RUN TRANSID</b> commands that have been issued by the user task.

Field name	Field ID	Description
ASFTCHCT	472	The number of <b>EXEC CICS FETCH CHILD</b> and <b>EXEC CICS FETCH ANY</b> commands that have been issued by the user task.
ASFREECT	473	The number of <b>EXEC CICS FREE CHILD</b> commands that have been issued by the user task.
ASFTCHWT	475	The elapsed time that the user task waited for a child task as a result of issuing an <b>EXEC CICS FETCH CHILD</b> or <b>EXEC CICS FETCH ANY</b> command that was not completed.
ASRNATWT	476	The elapsed time that the user task was delayed as a result of asynchronous child task limits managed by the asynchronous services domain.

Apart from the newly added monitoring fields **TRANFLAG** (field ID 164) in the **DFHTASK** group byte 4 has a hex value of **X'16'**, which indicates the task is started by the **EXEC CICS RUN TRANSID** command. For more information about **DFHTASK** see [IBM Knowledge Center](#).





## System tracking of asynchronous applications

You might want to track a business application for the purpose of billing, meeting service agreement, auditing, problem determination, or other services. CICS TS provides transaction tracking for you to determine the relationships between tasks in a region and across regions in a CICSplex. Transaction tracking can help you to see the tasks that are running in an application, and thus can help you to summarize the performance data for this group of related tasks, for auditing, problem determination, and other information gathering purposes.

If problems are seen in synchronous work then purging the transaction will remove the problematic transaction. However in asynchronous applications where parent and children tasks have different transactional scopes it can be important to work out if purging one task would affect other tasks. Using transaction tracking you can track down things such as a hanging parent task, working out whether cancelling a child task will affect any parent task, and the overall performance of the asynchronous applications etc.

This chapter discusses how tracking data can help you gather more information about your application. It also provides information about specific CICS commands that you can use to track tasks, to understand task relationships, and to monitor performance. This chapter includes the following topics:

- ▶ Data gathered by transaction tracking
- ▶ Using the INQUIRE ASSOCIATION command to track tasks
- ▶ Using CICS Explorer to track tasks
- ▶ Using CICS Performance Analyzer to understand task relationship
- ▶ Using IBM OMEGAMON for CICS on z/OS V5.5.0 to monitor performance

## 10.1 Data gathered by transaction tracking

Transaction tracking provides the capability to identify the relationships between tasks in an application as they flow across regions in a CICSplex. Transaction tracking records data and then passes the *association data* from the *origin task* to interrelated tasks. The association data carries the information for tracking these origin tasks and includes the following components:

- ▶ Origin data
- ▶ Previous transaction data
- ▶ Previous hop data
- ▶ Task context data
- ▶ Application context data

### 10.1.1 Origin data

*Origin data* tells you where the task was started and is created for those user tasks that are started by the following types of events:

- ▶ The **EXEC CICS START ATTACH** command
- ▶ The **EXEC CICS START** command with the `TERMINID` parameter specified
- ▶ A DTP or CPIC request
- ▶ A request over an APPC connection
- ▶ An HTTP request into CICS using CICS web support
- ▶ A request that is routed over an MRO connection to start a web service pipeline handler transaction
- ▶ A transaction start EP adapter
- ▶ A Java web application that runs in a Liberty JVM server
- ▶ The `CICS ExecutorService.runAsCICS()` method from a parent thread that is not running under a CICS task in Liberty JVM server
- ▶ The `CICS ExecutorService.runAsCICS()` method running OSGi JVM server

The complete list of [origin data characteristics](#) is in IBM Knowledge Center.

If you start your application using software other than CICS, using adapter data can help identify the transactions coming from that software. Adapter data is a part of origin data. You can use task-related user exit (TRUE) to add tracking information to the origin data. DFH\$APDT is a sample TRUE program showing how to add tracking information to the adapter data fields.

You can also add user correlation data with the XAPADMGR exit to add user information at the point of origin.

Origin data is passed to the next started tasks automatically in a region or between regions that use MRO or Internet Protocol interconnectivity (IPIC) connections.

You can find [examples of origin data creation](#) in IBM Knowledge Center.

Note that origin data is unrecoverable information, which means that the data is not available to any tasks that are attached because of a transaction restart or with any tasks that are rebuilt from the system log when a region is restarted.

## 10.1.2 Previous transaction data

*Previous transaction data* is available from CICS TS V5.4 or above. The data is created for tasks that are started locally by the **EXEC CICS RUN TRANSID** or **EXEC CICS START TRANSID** commands. The started task is not a new point of origin in order to qualify for adding the previous transaction data.

Previous transaction data tracks relationships within a single CICS region and provides the following information:

- ▶ The task in the same local region that requested the current task to be attached.
- ▶ The current task depth from one task to another in the same CICS region with which this task is associated. A value of zero indicates the task is the point of origin in the CICS system or is the first transaction that was the result of a request from one CICS system to another to initiate a task.

## 10.1.3 Previous hop data

If a request to attach a task is transmitted by using an IPIC or MRO connection between CICS TS 4.2 or later regions, the task that is attached as a result of this request has the *previous hop data* created by CICS. Previous hop data tracks tasks across interconnected regions, counting the hops between tasks as they flowed across regions.

Previous hop data describes the remote sender of the request so that the request can be tracked back into the previous CICS region. For example, it tells you the information of the previous CICS region, such as the network identifier (the APPLID), and the information of the previous task, such as the start time, its task number, transaction ID, and hop count.

You can find information about [previous hop data characteristics](#) in IBM Knowledge Center.

Previous transaction data, when combined with previous hop data, identifies both the local and remote sender of a request to attach a task and creates a trail that can be followed to the previous task or previous system, which enables data gathering to continue in the region that sent the request.

## 10.1.4 Task context data

*Task context data* is about information of the current task. Task context includes the APPLID of the CICS region, facility name and type associated with the task, IBM MVS™ image name, network name of the terminal started this task, user ID, start time of the task, unique transaction group ID, transaction ID of the task, first program of the task, and other relevant information. If the task is started from a TCP/IP client, you can get the TCPI/P client's IP address, port, and IP family as well.

For tasks started by TCPIP SERVICE socket, APPLDATA has the information of, for example inbound or outbound, the APPLID of the region this task runs in, the transaction ID that is defined for the TCPIP SERVICE, the network protocol, the TCPIP SERVICE name and IPCONN, and the partner region APPLID.

## 10.1.5 Application context data

*Application context data* is available from CICS TS 5.1 or later. You can use this data to determine the CICS application and platform information, if any, that is associated with the task. Application context data tells you the name of the application, the name of the platform

the application is deployed on, the version of the application, and the name of the operation. If the task is not associated with a CICS application, these fields are blank.

## 10.1.6 Flow of tracking data

Origin data, previous transaction data, and previous hop data are *flowed* from one task to the next when applicable. Figure 10-1 illustrates this flow. The scenario has both dynamic program link (DPL) and function shipping (FS) over MRO or IPIC connections.

As shown in Figure 10-1, both previous hop counts and previous transaction counts are represented with digits, with previous hop counts increasing horizontally between CICS regions and previous transaction counts increasing vertically down within a CICS region.

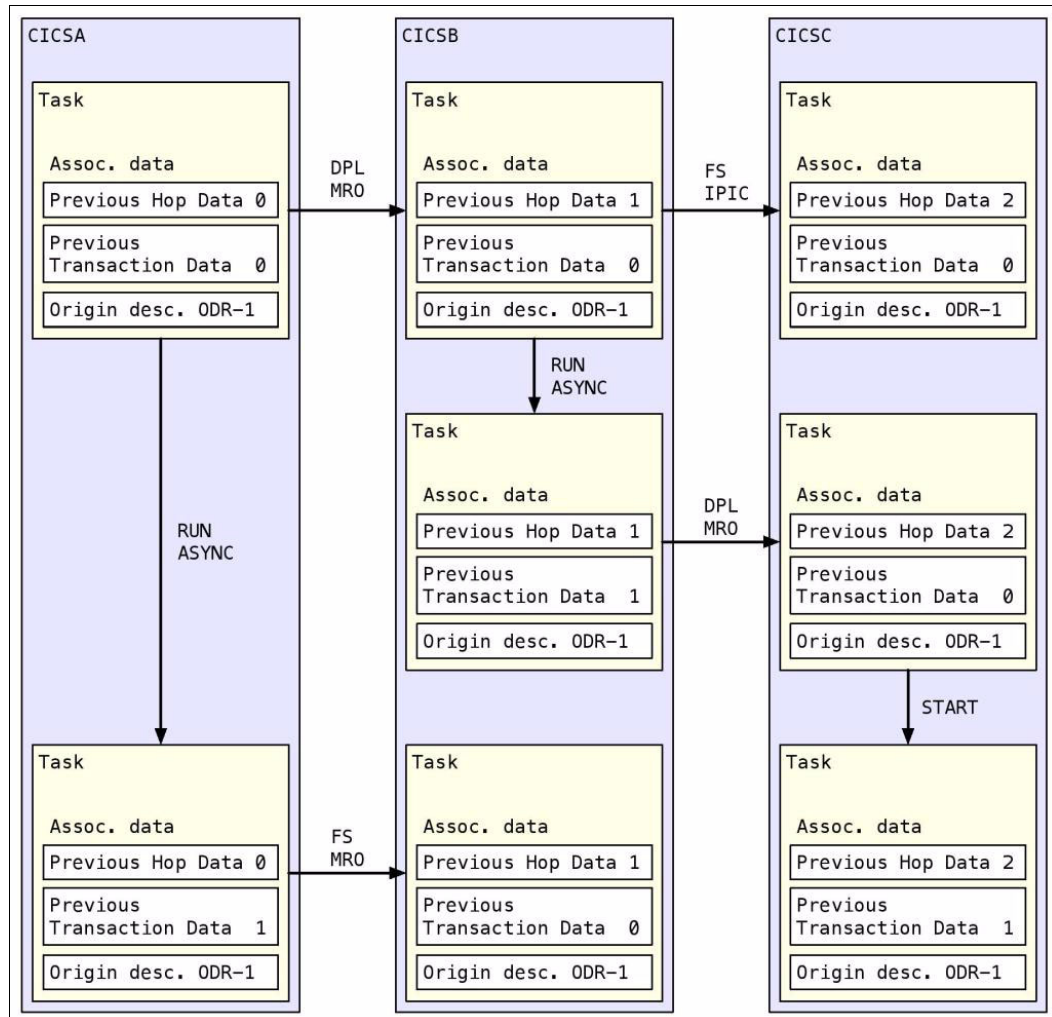


Figure 10-1 Example of tracking data flow from task to task

## 10.2 Using the INQUIRE ASSOCIATION command to track tasks

The **INQUIRE ASSOCIATION** command is a CICS system programming interface (SPI) command that you can use to retrieve association data for a running task, provided that you have the task number. CECI is a CICS supplied transaction. It can be used to quickly



determine the data that is associated with a certain task in the system by using the **INQUIRE ASSOCIATION** command.

For example, the following command returns association data for task 000073:

```
CECI INQUIRE ASSOCIATION(+0000073)
```

Example 10-1 shows the command output (with some extracted non-blank fields).

*Example 10-1 Output of CECI INQUIRE ASSOCIATION(+0000073)*

---

```
APPLId('IYCWZCAB') > current task's CICS region applid.
FACILType(+0000001209) > CVDA value for ASRUNTRAN which means this task is
started by RUN TRANSID.
ODAPplid('IYCWZCAB') > origin task's CICS region applid.
ODCLNTIpaddr('9.174.17.237')
ODCLNTPort(+0000053940)
ODFACILName('T130')
ODFACILType(+0000000213)
ODIpfamily(+0000000300)
ODLuname('IYCWT130')
ODNETId('GBIBMIYA')
ODNETWorkid('GBIBMIYA')
ODServerport(+0000000000)
ODStarttime('20170914090057.696545')
ODTaskid(+0000068) > task 000068 is the origin of the current task 0000073.
ODTRansid('PAS1')
ODUserid('HEJEN')
PHCount(+0000000000) > the current task is not in a chain of remote task as the
hop count is 0.
PRogram('JHEASCH1') > the program name of the current task.
PTCount(+0000000001) > the current task is started by a local task and has one
level of depth.
PTStarttime('20170914090057.696545') > Previous local task's start time. This is
the same as the origin task 0000068.
PTTaskid(+0000068) > the current task is started by local task 0000068.
PTTRansid('PAS1') > transaction ID of the local task 0000068
STarttime('20170914090218.224648') > current task's start time.
TRANsaction('HA41') > current task's transaction ID.
TRNgrpId('..GBIBMIYA.IYCWT130L..GF....') > the unique transaction group ID
which is the same as task 0000068.
USERId('HEJEN') > current task's user ID.
```

---

From these fields, you can determine that task 0000073 is a child task of 0000068 and was started by the **EXEC CICS RUN TRANSID** command. To determine information about the parent task 0000068, issue the following command:

```
CECI INQUIRE ASSOCIATION(+0000068)
```

Example 10-2 shows the output of the command (with some extracted non-blank fields). The output shows that origin data is created for this task and is passed to task 0000073.

*Example 10-2 Output of CECI INQUIRE ASSOCIATION( +0000068 )*

---

```
FACILName('T130') > this task is started from terminal T130.
FACILType(+0000000213) > CVDA value for TERMINAL
ODAPplid('IYCWZCAB')
ODCLNTIpaddr('9.174.17.237')
```

---

```

ODCLNTPort(+0000053940)
ODFACILName('T130 ')
ODFACILType(+0000000213)
ODIpfamily(+0000000300)
ODLuname('IYCWT130')
ODNETId('GBIBMIYA')
ODNETWorkid('GBIBMIYA')
ODServerport(+0000000000)
ODStartTime('20170914090057.696545')
ODTaskid(+00000068)
ODTCpips(' ')
ODTRansid('PAS1 ')
ODUserId('HEJEN ')
PHCount(+0000000000) > Previous hop count 0 means this task is not started from
a remote region.
Program('JHEASPG1')
PTCount(+0000000000) > Previous transaction count 0 means this task is not
started by a local task or is started by a local task with START ATTACH or START
TERMINAL.
TRANsaction('PAS1 ') > current task's transaction ID
TRNgrpId('..GBIBMIYA.IYCWT130L..GF....') > the unique transaction group ID which
is the same as task 0000073.
USERId('HEJEN ') > current task's user ID.

```

---

### 10.2.1 Building the picture of the application flow using the tracking data

Using the association information from the previous example, you can reverse engineer and build a picture that 0000068 is a parent task with the PAS1 transaction ID that started from a 3270 terminal, which in turn started a child task 0000073 with the HA41 transaction ID. This application flow is shown in Figure 10-2.

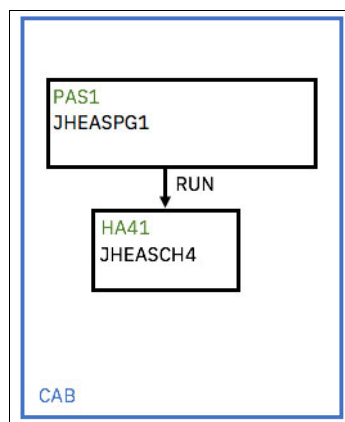


Figure 10-2 Application flow built from association data

## 10.3 Using CICS Explorer to track tasks

You can use the **INQUIRE ASSOCIATION** command to show a simple view of task flow. However, if the chain of the tasks is complicated or if you want a more intuitive way to determine the suspended task and its associated tasks, CICS Explorer can be helpful. CICS Explorer is a

system management tool based on Eclipse. It can connect to stand-alone CICS region or regions in CPSM.

You can find more information about [setting up access for CICS Explorer](#) in IBM Knowledge Center.

The following sections provide a high-level overview of CICS Explorer for the features related to tracking tasks.

### 10.3.1 Tracking interrelated tasks using search

You can see the running status of a task using the Tasks view in CICS Explorer. For example, Figure 10-3 shows the current tasks in a CICS region. A few of these are suspended. You can view the suspend reason by double-click a task, for example task 0000240, which opens the attributes view for this task as shown in Figure 10-4 on page 165.

Region	Task ID	Transaction ID	Run Status	User ID	Class Name
IYCWZCAB	0000240	PAS1	SUSPENDED	HEJEN	DFHTCL00
IYCWZCAB	0000256	CWWU	RUNNING	HEJEN	DFHTCL00
IYCWZCAB	0000243	HA41	SUSPENDED	HEJEN	DFHEDFTC
IYCWZCAB	0000254	CEDF	SUSPENDED	HEJEN	DFHTCL00
IYCWZCAB	0000251	CEDF	SUSPENDED	HEJEN	DFHTCL00

Figure 10-3 Tasks view in CICS Explorer

The suspend reason for this task is AS\_CHILD which means this task is trying to fetch the result from its asynchronous child task 0000243 but the child task hasn't finished (Figure 10-4).

Name	CICS Name	Value
SO Delay Time	SOMODDLY	0000:00:00.000000
Start Time	START	2017-11-02T12:46:18.342564+00:00
Stop Time	STOP	2017-11-02T12:47:20.778873+00:00
Suspend Reason	SUSPENDTYPE	AS_CHILD
Suspend Time	SUSPTIME	0000:01:02.435838
Suspended For Resource	SUSPENDVALUE	00000243

Figure 10-4 Task attributes view showing suspend reason

You might want to purge a suspended task. However before doing that, you can determine other tasks that are related to this task. To see the tasks that are related to a suspended task, or in fact any task, right-click a task row (in this example, task 0000075), and select **Search** → **Associated Tasks**, as shown in Figure 10-5 on page 166.

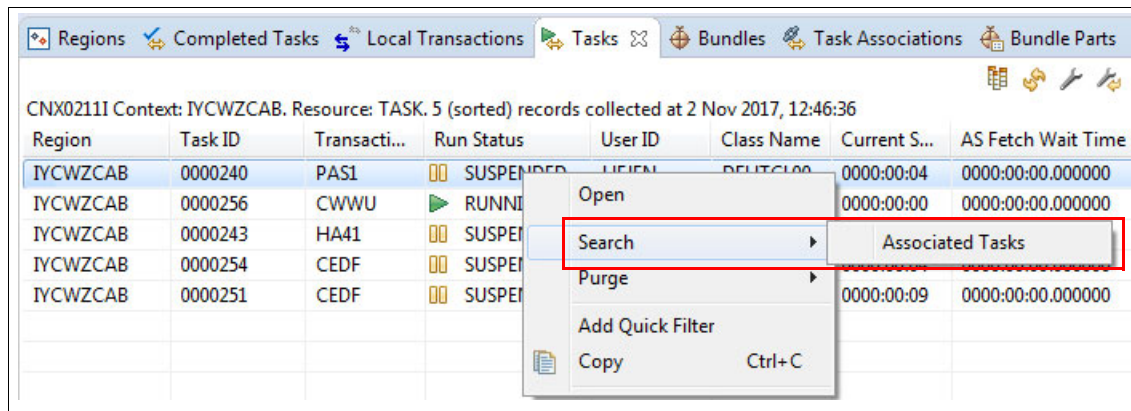


Figure 10-5 Search associated tasks menu in the Tasks view

The result of the search is shown in Figure 10-6, which shows that task 0000075 started task 0000078.

The screenshot shows the search results for task 0000240 in region IYCWZCAB. The table has columns: Tasks, Trans ID, Program, Prev Transaction Trans ID, Prev Transaction Task ID, Prev Hop Trans ID, and Prev Tr.

Tasks	Trans ID	Program	Prev Transaction Trans ID	Prev Transaction Task ID	Prev Hop Trans ID	Prev Tr
0000240	PAS1	JHEASPG1		0000000		0
0000243	HA41	JHEASCH4	PAS1	0000240		1

Figure 10-6 Search result of the associated tasks

### 10.3.2 Finding out associated tasks using the Task Associations views

To determine a list view of the association data information for all the tasks in the CICS region, open the Task Associations view as shown in Figure 10-7. You can see that task 0000138 has an Origin Transaction ID of PAS1, which is different from its own Trans ID of HA41 and which means that task 0000138 was started by another task. Task 0000135 displays in the Prev Transaction Task ID column.

The screenshot shows the 'Task Associations' view in CICS Explorer. The table has columns: Region, Task ID, Start Time, Trans ID, Origin Transaction ID, Origin App..., Prev Transaction Count, and Prev Transaction Task ID. Task 0000138 is highlighted with a red box.

Region	Task ID	Start Time	Trans ID	Origin Transaction ID	Origin App...	Prev Transaction Count	Prev Transaction Task ID
IYCWZCAB	0000135	2017-09-18T14:00:06.769700	PAS1	PAS1	IYCWZCAB	0	0000000
IYCWZCAB	0000138	2017-09-18T14:00:10.442360	HA41	PAS1	IYCWZCAB	1	0000135
IYCWZCAB	0000139	2017-09-18T14:00:10.442440	CEDF	CEDF	IYCWZCAB	0	0000000
IYCWZCAB	0000141	2017-09-18T14:00:10.450020	CEDF	CEDF	IYCWZCAB	0	0000000
IYCWZCAB	0000164	2017-09-18T14:09:57.325520	CWWU	CWWU	IYCWZCAB	0	0000000

Figure 10-7 Task Associations view in CICS Explorer

Double-click task 0000138 to show the details of the association data. Figure 10-8 shows that task 0000138 has a previous hop count of 0, a previous transaction count of 1, and the origin task is 0000135. This information indicates that task 0000138 as started by 0000135 in the same region.

Name	CICS Name	Value
Basic		
Current Application Context		
Distributed identity information		
Origin data		
Origin Adapter Data 1	ODAPTRDATA1	
Origin Adapter Data 2	ODAPTRDATA2	
Origin Adapter Data 3	ODAPTRDATA3	
Origin Adapter ID	ODAPTRID	
Origin Appl ID	ODAPPLID	IYCWCAB
Origin Appl ID Net ID	ODNETWORKID	GBIBMIYA
Origin Client IP Address	ODCLNTIPADDR	9.174.17.237
Origin Client Port	ODCLNTPORT	57932
Origin Facility Name	ODFACILNAME	T128
Origin Facility Type	ODFACILTYPE	TERMINAL
Origin IP Address Format	ODIPFAMILY	IPV4
Origin Net ID	ODNETID	GBIBMIYA
Origin Server Port	ODSERVERPORT	0
Origin Task	ODTASKID	0000135
Origin Task Start Date	ODSTARTTM	20170918140006.769703
Origin Task Start Time	ODSTARTTIME	2017-09-18T14:00:06.769703+00:00
Origin TCPIP Service	ODTCPIPS	
Origin Transaction ID	ODTRANSID	PAS1
Origin User ID	ODUSERID	HEJEN
Origin VTAM LU Name	ODLUNAME	IYCWT128
Previous hop data		
Prev Hop Appl Id	PHAPPLID	
Prev Hop Count	PHCOUNT	0
Prev Hop Net ID	PHNETWORKID	
Prev Hop Start Date	PHSTARTTM	
Prev Hop Start Time	PHSTARTTIME	0000-00-00T00:00:00.000000+00:00
Prev Hop Task ID	PHTASKID	0000000
Prev Hop Trans ID	PHTRANSID	
Previous transaction data		
Prev Transaction Count	PTCOUNT	1
Prev Transaction Start Date	PTSTARTTM	20170918140006.769703
Prev Transaction Start Time	PTSTARTTIME	2017-09-18T14:00:06.769703+00:00
Prev Transaction Task ID	PTTASKID	0000135
Prev Transaction Trans ID	PTTRANSID	PAS1

Figure 10-8 Task Association details for task 0000138 in CICS Explorer



As an example, double-click task 0000135 to show the details of the association data for this task. As shown in Figure 10-9 and Figure 10-10 on page 169, dozens of fields are available for this task, which is in-line with the information from the **INQUIRE ASSOCIATION** command.

Name	CICS Name	Value
▲ Basic		
Appl Data	APPLDATA	
Appl ID	APPLID	IYCWZCAB
CICS Release	EYU_CICSREL	E710
Client IP Address	CLIENTIPADDR	9.174.17.237
Client IP Format	CLNTIPFAMILY	IPV4
Client Port	CLIENTPORT	57932
Cluster Conn Type	CLIENTLOC	00000000000000000000000000000000
Facility Name	FACILNAME	T128
Facility Type	FACILTYPE	TERMINAL
IPCONN Resource	IPCONN	
MVS Image	MVSIMAGE	
Net ID	NETID	GBIBMIYA
Program	PROGRAM	JHEASPG1
Region	EYU_CICSNAME	IYCWZCAB
Server IP Address	SERVERIPADDR	0.0.0.0
Server IP Format	IPFAMILY	NOTAPPLIC
Server Port	SERVERPORT	0
Start Date	STARTTM	20170918140006.769703
Start Time	STARTTIME	2017-09-18T14:00:06.769703+00:00
Task ID	TASKID	0000135
TCP/IP Job	TCPIPJOB	
TCP/IP Security Zone	TCPIPZONE	
TCP/IP Service	TCPIPSERVICE	
Trans Group ID	TRNGRPID	1A11C7C2C9C2D4C9E8C14BC9E8C3E6E3F1F2F8D328771B304621A600
Trans ID	TRANSACTION	PAS1
User Correlation Data	USERCORRDATA	
User ID	USERID	HEJEN
VTAM LU Name	LUNAME	IYCWT128
▲ Current Application Context		
Application Major Version	ACMAJORVER	0
Application Micro Version	ACMICROVER	0
Application Minor Version	ACMINORVER	0
Application Name	ACAPPLNAME	
Application Operation Name	ACOPERNAME	
Application Platform Name	ACPLATNAME	
▲ Distributed identity information		
Distinguished Name	DNAME	
Realm	REALM	

Figure 10-9 Part 1 of Task Association details for task 0000135 in CICS Explorer

Figure 10-10 shows that task 0000135 has a previous hop count of 0 and a previous transaction count of 0, which indicate that this task is the origin.

Origin data		
Origin Adapter Data 1	ODAPTRDATA1	
Origin Adapter Data 2	ODAPTRDATA2	
Origin Adapter Data 3	ODAPTRDATA3	
Origin Adapter ID	ODAPTRID	
Origin Appl ID	ODAPPLID	IYCWZCAB
Origin Appl ID Net ID	ODNETWORKID	GBIBMIYA
Origin Client IP Address	ODCLNTIPADDR	9.174.17.237
Origin Client Port	ODCLNTPORT	57932
Origin Facility Name	ODFACILNAME	T128
Origin Facility Type	ODFACILTYPE	TERMINAL
Origin IP Address Format	ODIPFAMILY	IPV4
Origin Net ID	ODNETID	GBIBMIYA
Origin Server Port	ODSERVERPORT	0
Origin Task	ODTASKID	0000135
Origin Task Start Date	ODSTARTTM	20170918140006.769703
Origin Task Start Time	ODSTARTTIME	2017-09-18T14:00:06.769703+00:00
Origin TCPIP Service	ODTCPIPS	
Origin Transaction ID	ODTRANSID	PAS1
Origin User ID	ODUSERID	HEJEN
Origin VTAM LU Name	ODLUNAME	IYCWT128
Previous hop data		
Prev Hop Appl Id	PHAPPLID	
Prev Hop Count	PHCOUNT	0
Prev Hop Net ID	PHNETWORKID	
Prev Hop Start Date	PHSTARTTM	
Prev Hop Start Time	PHSTARTTIME	0000-00-00T00:00:00.000000+00:00
Prev Hop Task ID	PHTASKID	0000000
Prev Hop Trans ID	PHTRANSID	
Previous transaction data		
Prev Transaction Count	PTCOUNT	0
Prev Transaction Start Date	PTSTARTTM	
Prev Transaction Start Time	PTSTARTTIME	0000-00-00T00:00:00.000000+00:00
Prev Transaction Task ID	PTTASKID	0000000
Prev Transaction Trans ID	PTTRANSID	

Figure 10-10 Part 2 of Task Association details for task 0000135 in CICS Explorer

### 10.3.3 Graphical view of associated tasks

Without going through the fields, you can easily determine the relationship between tasks if you use the search function, as shown as in Figure 10-11. Right-click a task, and then select **Search** → **Associated Tasks**. (This menu is also accessible in the Tasks view.)

Region	Task ID	Start Time	Trans ID	Origin Transaction ID	Origin Ap...	Prev Transaction Count	Prev Transaction Task ID
IYCWZCAB	0000135	2017-09-18T14:00:06.769703	PAS1	PAS1	IYCWZCAB	0	0000000
IYCWZCAB	0000138	Open	A41	PAS1	IYCWZCAB	1	0000135
IYCWZCAB	0000139				ZCAB	0	0000000
IYCWZCAB	0000141				ZCAB	0	0000000
IYCWZCAB	0000164					0	0000000

Figure 10-11 Menu for searching Associated Task in CICS Explorer

The result of the search displays in a Search view, as shown in Figure 10-12. Task 0000135 is at the top level, with task 0000138 being a child of it. This tree view is the same as the application flow shown in Figure 10-2 on page 164, which was reversed engineered using the **INQUIRE ASSOCIATION** command.

Tasks associated with task "0000135" in region "IYCWCZCAB" - 2 results - 16:53:54

Tasks	Trans ID	Program	Prev Transaction Tr...	Prev Transaction Task ID	Prev Transaction Count	Appl ID	Prev Hop Task ID	Prev Hop Trans ID
0000135	PAS1	JHEASPG1		0000000	0	IYCWCZCAB	0000000	
0000138	HA41	JHEASCH4	PAS1	0000135	1	IYCWCZCAB	0000000	

Figure 10-12 Search result for associated tasks in CICS Explorer

### 10.3.4 Graphical view of orphaned tasks

Figure 10-13 shows an example where there are three tasks with the same origin transaction in the same CICS region. In this example, the origin transaction CECI started PAS1, which completes the **RUN TRANSID** command to start HA41.

CNX02111 Context: IYCWCZCAB. Resource: TASKASSC. 6 records collected at 19 Sep 2017, 13:56:26

ion	Task ID	Start Time	Trans ID	Origin Transaction ID	Origin Ap...	Prev Transaction Count	Prev Transaction Task ID
WZCAB	0000146	2017-09-19T12:42:00.629493+00:00	CECI	CECI	IYCWCZCAB	0	0000000
WZCAB	0000233	2017-09-19T12:56:17.985390+00:00	PAS1	CECI	IYCWCZCAB	1	0000146
WZCAB	0000238	2017-09-19T12:56:22.352011+00:00	HA41	CECI	IYCWCZCAB	2	0000233

Figure 10-13 View of three tasks with the same origin transaction in CICS Explorer

A search for any of these three tasks shows the tree view shown in Figure 10-14. It is clear that task 0000146 at the top row started task 0000233, which in turn started child task 0000238.

Tasks associated with task "0000176" in region "IYCWCZCAB" - 3 results - 13:56:29

Tasks	Trans ID	Program	Prev Transaction Ta...	Prev Transaction C...	Prev Transaction Tr...	Appl ID	Facility Type	Suspend Reason
0000146	CECI	DFHECIP	0000000	0		IYCWCZCAB	TERMINAL	ZCLOWAIT
0000233	PAS1	JHEASPG1	0000146	1	CECI	IYCWCZCAB	START	EDF
0000238	HA41	JHEASCH4	0000233	2	PAS1	IYCWCZCAB	ASRUNTRAN	EDF

Figure 10-14 Tree view of three associated tasks in CICS Explorer

If the middle task 0000233 has finished, refreshing the Search view reflects those tasks that are live in the CICS region. Figure 10-15 shows the refreshed view.

Tasks associated with task "0000238" in region "IYCWCZCAB" - 1 result - 13:59:24

Tasks	Trans ID	Program	Prev Transaction Ta...	Prev Transaction C...	Prev Transaction Tr...	Appl ID	Facility Type	Suspend Reason
0000146	CECI					IYCWCZCAB	TERMINAL	
Orphaned Tasks								
0000233	PAS1					IYCWCZCAB		
0000238	HA41	JHEASCH4	0000233	2	PAS1	IYCWCZCAB	ASRUNTRAN	EDF

Figure 10-15 Tree view of orphaned task in CICS Explorer



Because task 0000233 is finished, its child task 0000238 became an orphaned task and is gathered under the Orphaned Tasks section. From task 0000238's previous transaction data, the search view shows 0000233 is the previous transaction task number. However, the system can't get the association data for task 0000233 because it has already completed. Therefore the search view doesn't have the information regarding task 0000233 and cannot reliably trace back to task 0000146. The search view knows that task 0000146 is related to task 0000238 as the origin, so task 0000146 is shown with limited association information.

## 10.4 Using CICS Performance Analyzer to understand task relationship

Using the **INQUIRE ASSOCIATION** command and the Task Associations view in CICS Explorer are good for real-time viewing and diagnosing tasks. For purposes such as planning, auditing, tuning of CICS workload, CICS monitoring data which is available offline has a rich set of information for these purposes.

The DFHCICS group in the performance class of CICS monitoring contains origin data information, previous hop data, and previous transaction data for completed tasks. This section uses CICS Performance Analyzer (CICS PA) to track task relationships.

### 10.4.1 Brief overview of CICS Performance Analyzer

[CICS Performance Analyzer](#) is a reporting tool that provides reports and analysis about the performance of your CICS systems and applications. With these reports, it helps you tune, manage, and plan your CICS systems in an efficient way. CICS Performance Analyzer is designed to complement the CICS-supplied utilities and sample programs for monitoring and statistics, such as DFH\$MOLS and DFHSTUP.

The CICS Performance Analyzer reports are based on input data that is collected in system management facility (SMF) data. CICS Performance Analyzer provides reporting and analysis for the following SMF data:

- ▶ CICS Monitoring Facility (CMF) performance, exception, and transaction resource class records (type 110 subtype 1)
- ▶ CICS statistics and server statistics data (type 110 subtype 2)
- ▶ CICS Transaction Gateway statistics (type 111)
- ▶ DB2 accounting records (type 101)
- ▶ WebSphere MQ accounting data (type 116)
- ▶ System Logger data (type 88)
- ▶ IBM OMEGAMON® for CICS data (type 112)

To generate CICS monitoring data to SMF, you need to [switch on CICS monitoring](#) for the region and select which class of monitoring data to be collected.

You can find a rich set of scenarios to get you started with CICS PA in *CICS Performance Analyzer*, [SG24-6063](#). For the latest [CICS Performance Analyzer documentation](#), visit IBM Knowledge Center.

## 10.4.2 Extending the business application

This section describes an extension to the scenario shown in Figure 10-2 on page 164 to a simple business application. This business application is across two systems, completes a dynamic program link (DPL) from one system to another, and starts a few child transactions using the **RUN TRANSID** command. Figure 10-16 illustrates the application flow.

Transaction PG1 starts two local transactions, HA41 and HA11. Transaction HA11 does a DPL call over an IPIC connection to start the remote program JHEASPG4. This remote program starts a local transaction HA31, and transaction HA31 starts another local transaction HA41.

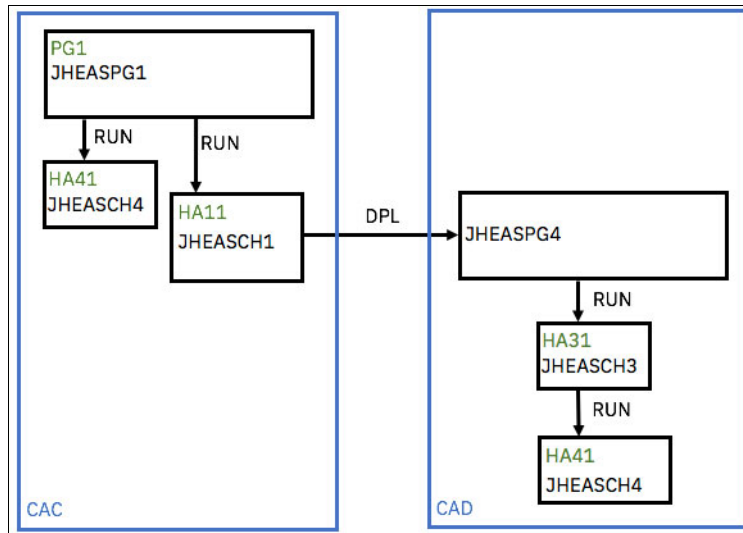


Figure 10-16 A simple application cross systems and starting local transactions

Although the application is not complicated, there are six tasks involved in this application. Some of the tasks might not display when you view the associated tasks in real-time in the CICS Explorer. As shown in Figure 10-17, the task for transaction HA41 in system CAC has completed and does not display, so the view isn't a complete view.

Task ID	Trans ID	Prev Hop Trans ID	Prev Transaction Trans ID	Program	Prev Hop Task ID	Prev Transaction Ta...	App
0000074	PG1			JHEASPG1	0000000	0000000	IYCV
0000080	HA11		PG1	JHEASCH1	0000000	0000074	IYCV
0000091	CSMI	HA11		DFHMIRS	0000080	0000000	IYCV
0000092	HA31		CSMI	JHEASCH3	0000000	0000091	IYCV
000121	HA41		HA31	JHEASCH4	0000000	0000092	IYCV

Figure 10-17 Task tree view for cross-system application using the CICS Explorer

You can thus use CICS Performance Analyzer to look at the CICS-collected monitoring data for this application.

## 10.4.3 Transaction tracking reports by CICS Performance Analyzer

CICS Performance Analyzer provides a sample [Transaction Tracking List report](#) and [Transaction Tracking Summary report](#) in the performance reports category. These reports use

the transaction group ID to identify related transactions, whether they are on the same CICS system or on different CICS systems, and to produce the relationship between them.

We dumped the SMF data for both regions and used the Transaction Tracking List report to show the tasks in time sequence. Figure 10-18 shows one section from the output of the report by the sequence of the time when task is started.

At the top of the section, it shows the origin information. The originating transaction is PG1, from the CICS region IYCWZCAC, the task number of 74, which was started from a terminal called TC45.

Next is a list of subordinate transactions from the origin transaction, which can be previous hop transactions in a remote region or previous transactions in the same region. This list shows the HA41 transaction with task number 77, which was missing from the previous live view shown in Figure 10-17 on page 172. This list matches the application shown in Figure 10-16 on page 172.

The first column in Figure 10-18 is the transaction ID (Tran), followed by PTTTran (previous transaction ID in the same region), and then PHTran (previous hop transaction ID in the remote region). One transaction cannot have both PTTTran and PHTran as their immediate previous transaction. Either one of these two columns has a value or none, thereby indicating an origin task.

The PRTaskNo column shows the task number of either previous transaction, previous hop transaction, or 0.

The PRCount column shows the previous transaction count or the previous hop count if the previous transaction count is 0.

V5R4M0														
CICS Performance Analyzer														
Transaction Tracking List - Call Sequence														
TTLS0001 Printed at 16:02:37 9/22/2017										Data from 10:09:17 9/22/2017 to 12:06:24 9/22/2017				
													Page	4
OTran	OUserid	OAPPLID	OTaskNo	OStart	OOrigin	OFcty	OTCIPSP	OCli6Adr	OCLIPORT					
PG1	CICSUSER	IYCWZCAC	74	11:57:12.3034	TERM	TC45		9.174.17.237	55361					
Tran	PTTran	PHTran	TaskNo	PRTaskNo	PRCount	Origin	APPLID	PHAPPLID	Start	Response	User	CPU	Suspend	OSLatncy
									Time	Time	Time	Time	Time	Time
PG1			74	0	0	TERM	IYCWZCAC		11:57:12.3034	496.3656	.0022	496.3535	.0000	.0000
HA41	PG1		77	74	1	ASRUN	IYCWZCAC		11:57:19.8965	.0066	.0010	.0001	7.5931	7.5931
HA11	PG1		80	74	1	ASRUN	IYCWZCAC		11:57:22.9428	475.5003	.0016	475.4926	10.6394	10.6394
CSMI		HA11	91	80	1	IPIC	IYCWZCAD	IYCWZCAC	11:57:47.0377	451.4053	.0004	451.4049	34.7344	24.0949
HA31	CSMI		92	91	1	ASRUN	IYCWZCAD		11:57:47.0379	444.2284	.0015	444.2251	34.7346	.0002
HA41	HA31		121	92	2	ASRUN	IYCWZCAD		11:59:35.9804	360.0740	.0003	360.0736	143.6770	108.9425

Figure 10-18 Transaction Tracking List report according to time sequence

You can also generate a call sequence report. Using the call sequence report, you can easily work out the flow of the tasks and where the flow ends. For the sample shown in Figure 10-19, there are two subsequences from transaction PG1. The first subsequence is PG1 to HA41, and the second is PG1 to HA11 to CSMI to HA31 then to HA41.

CICS Performance Analyzer															
Transaction Tracking List - Time Sequence															
TTL50001 Printed at 12:56:57 9/22/2017						Data from 10:09:17 9/22/2017 to 12:06:24 9/22/2017						Page 5			
-----															
OTran	OUserid	OAPPLID	OTaskNo	OStart	OOrigin	OFcty	OTCPIPSr	OCLi6Adr	OCLIPORT						
				Time											
PG1	CICSUSER	IYCWCZC	74	11:57:12.3034	TERM	TC45		9.174.17.237	55361						
Tran	PTTran	PHTran	TaskNo	PRTaskNo	PRCount	Origin	APPLID	PHAPPLID	Start	Response	User	CPU	Suspend	OSLatncy	PRLatncy
									Time	Time	Time	Time	Time	Time	Time
PG1			74	0	0	TERM	IYCWCZC		11:57:12.3034	496.3656	.0022	496.3535	.0000	.0000	.0000
HA41 PG1			77	74	1	ASRUN	IYCWCZC		11:57:19.8965	.0066	.0010	.0001	7.5931	7.5931	
HA11 PG1			80	74	1	ASRUN	IYCWCZC		11:57:22.9428	475.5003	.0016	475.4926	10.6394	10.6394	
CSMI	HA11		91	80	1	IPIC	IYCWCZC	IYCWCZC	11:57:47.0377	451.4053	.0004	451.4049	34.7344	24.0949	
HA31	CSMI		92	91	1	ASRUN	IYCWCZC		11:57:47.0379	444.2284	.0015	444.2251	34.7346	.0002	
HA41 HA31			121	92	2	ASRUN	IYCWCZC		11:59:35.9804	360.0740	.0003	360.0736	143.6770	108.9425	
-----															

Figure 10-19 Transaction Tracking List report according to call sequence

The Performance Transaction Tracking Summary report displays a summary of all transactions that are associated with an originating transaction. Eight instances of transaction PG1 ran, as shown in Figure 10-20.

CICS Performance Analyzer Performance Transaction Tracking Summary															
V5R4M0				CICS Performance Analyzer Performance Transaction Tracking Summary											
TTSU0001 Printed at 12:56:57 9/22/2017				Data from 10:09:17 9/22/2017 to 12:06:24 9/22/2017								Page 1			
PHAPPLID	PRTran	PRCount	APPLID	Tran	PR %	#Tasks	Avg Response Time	Max Response Time	Avg Dispatch Time	Avg User Time	Avg CPU Time	Avg Suspend Time	Max Suspend Time	Avg DispWait Time	Avg FC Wait Time
	CSMI	0	IYCWCZC	PG1		8	64.2596	496.3656	.0044	.0008	64.2552	496.3535	.0003	.0000	
	PG1	1	IYCWCZC	HA31	75	6	74.0381	444.2284	.0006	.0003	74.0375	444.2251	.0002	.0000	
	PG1	1	IYCWCZC	HA11	100	8	59.7918	475.5003	.0099	.0038	59.7819	475.4926	.0004	.0000	
	PG1	1	IYCWCZC	HA41	100	8	.0009	.0066	.0009	.0002	.0000	.0001	.0000	.0000	
IYCWCZC	HA11	1	IYCWCZC	CSMI	75	6	75.2347	451.4053	.0003	.0003	75.2344	451.4049	.0001	.0000	
	HA31	2	IYCWCZC	HA41	75	6	60.0124	360.0740	.0001	.0001	60.0123	360.0736	.0001	.0000	
Total						42	53.5269	496.3656	.0030	.0010	53.5239	496.3535	.0002	.0000	

Figure 10-20 Performance Transaction Tracking Summary report

If any of the tasks are missing from unloaded SMF data, for reasons such as the record was not collected or unloaded, a [Transaction Tracking List—Recap Report](#) is created to show the number of originating tasks that have children transactions missing and the number of children transactions that don't have parent transactions.

#### 10.4.4 Transaction group reports by CICS Performance Analyzer

CICS Performance Analyzer has a dedicated transaction group report. Transactions are grouped by the transaction group ID, which was assigned when the originating transaction was started.

Figure 10-21 shows the equivalent Transaction Group report for the flow of tasks shown in the Transaction Tracking List reports in Figure 10-18 on page 173 and Figure 10-19 on page 174.

CICS Performance Analyzer															
Transaction Group															
V5R4M0															
TRGP0001 Printed at 12:56:57 9/22/2017 Data from 10:09:17 9/22/2017 to 12:06:24 9/22/2017															
														Page	1
Tran	Userid	SC	Origin	Brdg Tran	Client IP Address	Request Type	Program	Term	LUName	Fcty T/Name	Conn Name	APPLID	R Task T	Stop Time	Response Time
HA41	CICSUSER	U	ASRUN			AP:	JHEASCH4					IYCWZCAD	121 T	12:05:36.05	360.074
PG1	CICSUSER	TO	TERMINAL		9.174.17.237	AP:	JHEASPG1	TC45	IYCWTC45	D/TC45		IYCWZCAC	74 T	12:05:28.67	496.365
HA11	CICSUSER	U	ASRUN			AP:	JHEASCH1					IYCWZCAC	80 T	12:05:18.44	475.500
CSM1	CICSUSER	U	IPIC SESS		9.20.5.0	AP:----	JHEASPG4					IYCWZCAD	91 T	12:05:18.44	451.405
HA31	CICSUSER	U	ASRUN			AP:	JHEASCH3					IYCWZCAD	92 T	12:05:11.27	444.228
HA41	CICSUSER	U	ASRUN			AP:	JHEASCH4					IYCWZCAC	77 T	11:57:19.90	.0066

Figure 10-21 Transaction Group report

The Transaction Group Summary report shown in Figure 10-22 provides the number of transactions for a particular type of origin and their average performance figures, such as response time, dispatch time, CPU time, and so on. This type of report produces similar measurement fields as the Performance Transaction Tracking Summary report but from a different angle.

V5R4M0		CICS Performance Analyzer Transaction Group - Summary									
TRGP0001 Printed at 12:56:57		9/22/2017 Data from 10:09:17		9/22/2017 to 12:06:24		9/22/2017		Page	3		
Origin Type	Transactions	Average Response	Average Dispatch	Average CPU Time	Average Suspend	Average DispWait	Average IR Wait	Average RMI Susp	Average FC Wait	Average SO Wait	
ASRUN	35	36.648	.000	.000	.000	.000	.000	.000	.000	.000	
IPIC SESS	12	37.898	.000	.000	.000	.000	.000	.000	.000	.000	
NONE	191	367.733	.000	.000	.000	.000	.000	.000	.000	.000	
START	19	.013	.000	.000	.000	.000	.000	.000	.000	.000	
TERM START	35	23.943	.000	.000	.000	.000	.000	.000	.000	.000	
TERMINAL	99	63.530	.000	.000	.000	.000	.000	.000	.000	.000	
WEB	83	.006	.000	.000	.000	.000	.000	.000	.000	.000	
-----		-----		-----		-----		-----		-----	
TOTAL	474	166.883	.000	.000	.000	.000	.000	.000	.000	.000	

Figure 10-22 Transaction Group - Summary report

## 10.5 Using IBM OMEGAMON for CICS on z/OS V5.5.0 to monitor performance

[IBM OMEGAMON for CICS](#) on z/OS V5.5.0 is part of the OMEGAMON family of products that provides systems monitoring and management of IBM Z systems. It centrally monitors and manages CICS resources, transactions, and interactions with other subsystems, such as IBM DB2 and IBM Information Management System (IMS™). It can monitor and trace real-time task information and keep historical data for trend analysis over a period of time.

This section shows a scenario where alerts show up in OMEGAMON and then can be diagnosed in OMEGAMON.

## 10.5.1 Alerts showing up in OMEGAMON

Figure 10-23 shows the Service Level Summary workspace within the OMEGAMON Enhanced 3270 user interface. The data shown in this workspace provides performance information about each defined service class grouping relevant to their defined goal for a CICSplex that is monitored by the OMEGAMON for CICS agent. In this example, the top two service classes show a warning on the average response time. To determine the problem, you can drill down on the service class by type S (for example, GTRANS).

File Edit View Tools Navigate Help 09/21/2017 08:58:43										
Command ==> CICSplex Service Level Summary										
CICSplex Service Level Analysis for OMEGPLEX										
Columns 2 to 11 of 39 Rows 1 to 1										
Service Class Name	Workload Name	ΔAverage Response Time	Transactions Total	ΔPerformance Index	Transaction Rate	Interval End Timestamp	Average time using CPU	Highest Response Time	Highest Time Using CPU	Highest T
STRANS	DFLTWORK	30.6131s	2	30.61%	2	08:58:37	.001723s	1m.01s	.003414s	
GTRANS	DFLTWORK	28.3841s	6	28.38%	6	08:58:37	.000170s	50.0467s	.000412s	
KTRANS	DFLTWORK	.001572s	4	0.00%	4	08:58:37	.000044s	.002348s	.000046s	
ITRANS	DFLTWORK	.002086s	1	0.00%	1	08:58:37	.000045s	.002086s	.000045s	
WTRANS	DFLTWORK	.001857s	3	0.00%	3	08:58:37	.000044s	.002045s	.000045s	
FTRANS	DFLTWORK	.001385s	2	0.00%	2	08:58:37	.000045s	.001947s	.000046s	
XTRANS	DFLTWORK	.001322s	2	0.00%	2	08:58:37	.000044s	.001821s	.000047s	
UTRANS	DFLTWORK	.001781s	2	0.00%	2	08:58:37	.000044s	.001848s	.000045s	
HTRANS	DFLTWORK	.001584s	1	0.00%	1	08:58:37	.000044s	.001584s	.000044s	
OTRANS	DFLTWORK	.001506s	2	0.00%	2	08:58:37	.000043s	.001545s	.000044s	
PTRANS	DFLTWORK	.001733s	3	0.00%	3	08:58:37	.000044s	.001969s	.000045s	
TTRANS	DFLTWORK	.001429s	1	0.00%	1	08:58:37	.000044s	.001429s	.000044s	
NTRANS	DFLTWORK	.001322s	1	0.00%	1	08:58:37	.000044s	.001322s	.000044s	
ATTRANS	DFLTWORK	.001277s	3	0.00%	3	08:58:37	.000043s	.001804s	.000044s	
ETTRANS	DFLTWORK	.001110s	3	0.00%	3	08:58:37	.000044s	.001191s	.000045s	
QTRANS	DFLTWORK	.001062s	1	0.00%	1	08:58:37	.000045s	.001062s	.000045s	
MTRANS	DFLTWORK	.001710s	3	0.00%	3	08:58:37	.000053s	.002247s	.000070s	
ZTRANS	DFLTWORK	.000860s	2	0.00%	2	08:58:37	.000103s	.001020s	.000153s	
RTRANS	DFLTWORK	.002049s	1	0.00%	1	08:58:37	.000044s	.002049s	.000044s	
VTRANS	DFLTWORK	.000903s	1	0.00%	1	08:58:37	.000044s	.000903s	.000044s	
LTRANS	DFLTWORK	.000625s	1	0.00%	1	08:58:37	.000161s	.000625s	.000161s	

Figure 10-23 Service Level Summary provided by OMEGAMON for CICS on z/OS

Figure 10-24 shows the details of the selected service class, that is the transaction IDs in this class. You can select the GBPA transaction ID to see this transaction's details, which are shown in Figure 10-25 on page 177. Figure 10-24 shows that GBPA has a response time goal of an average of 1 second.

File Edit View Tools Navigate Help 09/21/2017 08:58:53										
Command ==> CICSplex Service Class Detail										
CICSplex Transactions for Service Class GTRANS										
Columns 2 to 10 of 38 Rows 1 to 1										
Transaction ID	ΔAverage Response Time	Transactions Total	ΔPerformance Index	Transaction Rate	Interval End Timestamp	Average time using CPU	Highest Response Time	Highest Time Using CPU	Number of Abended	
GBPA	50.0467s	1	50.04%	1	08:58:37	.000412s	50.0467s	.000412s		
GCD3	50.0463s	1	50.04%	1	08:58:37	.000172s	50.0463s	.000172s		
GCD2	40.0849s	1	40.08%	1	08:58:37	.000166s	40.0849s	.000166s		
GCD1	30.1234s	1	30.12%	1	08:58:37	.000187s	30.1234s	.000187s		
GEA4	.001109s	1	0.00%	1	08:58:37	.000044s	.001109s	.000044s		
GEA3	.002010s	1	0.00%	1	08:58:37	.000044s	.002010s	.000044s		
CICSplex Regions for Service Class GTRANS										
Columns 2 to 10 of 38 Rows 1 to 1										
CICS Region Name	ΔAverage Response Time	Transactions Total	ΔPerformance Index	Transaction Rate	Interval End Timestamp	Average time using CPU	Highest Response Time	Highest Time Using CPU	Number of Abended	
CICSR54L	28.3841s	6	28.38%	6	08:58:37	.000170s	50.0467s	.000412s		

Figure 10-24 Service Class Detail provided by OMEGAMON for CICS on z/OS



Figure 10-25 shows that GBPA has a response time goal of an average of 1 second, but the actual average response time is over 50 seconds. Thus, the alert is raised.

File Edit View Tools Navigate Help 09/21/2017 08:59:05				Auto Update : Off	
Command ==> KCPPSLDT				CICSplex : OMEGPLEX	
Service Class Transaction Detail					
Service Class Detail for GTRANS Transaction GBPA					
■					
Average Response Time.....		50.00467s	Average time using CPU.....		.000412s
Highest Response Time.....		50.00467s	Highest Time Using CPU.....		.000412s
Resp Time Std Deviation.....		0.00000s	CPU Time Std Deviation.....		0.00000s
Transactions Total.....		1	Transaction Rate.....		1
Average DSA Occupancy.....		0K	Average EDSA Occupancy.....		2K
Performance Index.....		50.00%	Goal Type.....		Average
Percent of Goal.....		0%	Response Time Goal.....		1.000s
Interval Length.....		1m 00s	Interval Start Timestamp.....		08:57:37
Interval End Timestamp.....		08:58:37	Transaction Abends.....		0
■					
Transaction Time Averages					
Time Using CPU.....		.000412s	Wait on IC Delay.....		0.00000s
Wait on MRO.....		0.00000s	Wait on Enq Delay.....		0.00000s
Wait on Journal Control.....		0.00000s	Dispatch Time.....		.000437s
Wait on 1st dispatch.....		.000011s	Wait on Redispach.....		0.00000s
Wait on TD IO.....		0.00000s	Wait on TS IO.....		0.00000s
Wait on MQ.....		0.00000s	Wait on File requests.....		0.00000s
Time Using RLS CPU.....		0.00000s	Unidentifiable wait.....		50.0460s
Time in DB2.....		0.00000s	Wait on DLI.....		0.00000s
Time in ADABAS.....		0.00000s	Time in IDMS.....		0.00000s
Time in Datacom.....		0.00000s	Time in SUPRA.....		0.00000s
■					
Time Distribution Detail					
% Time Using CPU.....		0%	% Wait on IC Delay.....		0%
% Wait on MRO.....		0%	% Wait on Enq Delay.....		0%
% Wait on Journal Control.....		0%	% of time Dispatched.....		0%
% Wait for 1st Dispatch.....		0%	% Wait on Redispach.....		0%
% Wait on TD IO.....		0%	% Wait on TS IO.....		0%
% Wait on MQ.....		0%	% Wait on File Control.....		0%
% Time Using RLS CPU.....		0%	% Wait on unidentifiable.....		99%
% Wait on DB2.....		0%	% Wait on DLI.....		0%
% Wait on ADABAS.....		0%	% Wait on IDMS.....		0%
% Wait on Datacom.....		0%	% Wait on SUPRA.....		0%
■					
Response Distribution Detail					
50% of Goal.....		0	60% of Goal.....		0
70% of Goal.....		0	80% of Goal.....		0
90% of Goal.....		0	100% of Goal.....		0
110% of Goal.....		0	120% of Goal.....		0
130% of Goal.....		0	140% of Goal.....		0
150% of Goal.....		0	200% of Goal.....		0
400% of Goal.....		0	> 400% of Goal.....		1

Figure 10-25 Transaction details within a service class provided by OMEGAMON for CICS on z/OS

## 10.5.2 Drill down to the problematic task

To look at the task details for the highest response time task, position your cursor on the Highest Response Time, and press Enter. Figure 10-26 shows the task details from the historical data that was gathered, for example the task number, the program name, CPU time, user ID, response time, and so forth.

File Edit View Tools Navigate Help 09/21/2017 10:51:55		Display : HISTORY																																				
Command ==> KCPTASHE		CICSplex : OMEGPLEX																																				
Task History Detail		Region : CICSRS54L																																				
Details Statistics Storage Timings Related																																						
Task History Detail for Task Number 11992																																						
<table><tr><td>Transaction ID.....</td><td>GBPA</td><td>CPU Time.....</td><td>.000431s</td></tr><tr><td>Response Time.....</td><td>50.2119s</td><td>Task Number.....</td><td>11992</td></tr><tr><td>End Time.....</td><td>08:48:41</td><td>Start Time.....</td><td>08:45:51</td></tr><tr><td>User ID.....</td><td>CICSR54L</td><td>Program ID.....</td><td>GBPARENT</td></tr><tr><td>Storage HWM.....</td><td>3936</td><td>File Requests.....</td><td>0</td></tr><tr><td>Terminal ID.....</td><td>H437</td><td>Terminal I/O.....</td><td>34</td></tr><tr><td>ABEND Code.....</td><td></td><td>Trace active.....</td><td>Yes</td></tr><tr><td>End Date.....</td><td>09/21/17</td><td>Start Date.....</td><td>09/21/17</td></tr><tr><td>Asynchronous Status.....</td><td>Parent</td><td></td><td></td></tr></table>			Transaction ID.....	GBPA	CPU Time.....	.000431s	Response Time.....	50.2119s	Task Number.....	11992	End Time.....	08:48:41	Start Time.....	08:45:51	User ID.....	CICSR54L	Program ID.....	GBPARENT	Storage HWM.....	3936	File Requests.....	0	Terminal ID.....	H437	Terminal I/O.....	34	ABEND Code.....		Trace active.....	Yes	End Date.....	09/21/17	Start Date.....	09/21/17	Asynchronous Status.....	Parent		
Transaction ID.....	GBPA	CPU Time.....	.000431s																																			
Response Time.....	50.2119s	Task Number.....	11992																																			
End Time.....	08:48:41	Start Time.....	08:45:51																																			
User ID.....	CICSR54L	Program ID.....	GBPARENT																																			
Storage HWM.....	3936	File Requests.....	0																																			
Terminal ID.....	H437	Terminal I/O.....	34																																			
ABEND Code.....		Trace active.....	Yes																																			
End Date.....	09/21/17	Start Date.....	09/21/17																																			
Asynchronous Status.....	Parent																																					

Figure 10-26 Task History Detail monitored by OMEGAMON for CICS on z/OS

There are four other tabs that shown information related to this task. To see the related tasks, go to the last tab called *Related*.

Figure 10-27 shows the found related tasks and how they are related together. As shown in Figure 10-27, GBPA is related to GCD1, GCD2, GCD3, and GCD4 with asynchronous child and parent relationships.

File Edit View Tools Navigate Help 09/21/2017 08:59:34

Command ==> KCPTASHL

Task History Detail

Display : HISTORY  
CICSplex : OMEGPLEX  
Region : CICSRESL

Details

Statistics

Storage

Timings

Related

Asynchronous Transactions related to this Task

Columns 2 to 8 of 8

Rows 1 to 5 of 5

Transaction ID	Asynchronous Status	CPU Time	Overall Elapsed Time MS	Total Wait Time	Dispatch Time	Task Number	Task Status
- GBPA	Parent	.000431s	50.2119s	50.211s	0.000s	11992	Done
- GCD3	Child	.000163s	50.2114s	50.211s	0.000s	11995	Done
- GCD2	Child	.000243s	40.2500s	40.249s	0.000s	11994	Done
- GCD1	Child	.000257s	30.0266s	30.026s	0.000s	11993	Done
- GCD4	Child	2.58363s	10.8906s	8.161s	2.728s	11996	Done

No related tasks were found for this unit of work

Figure 10-27 Related tasks for a certain task monitored by OMEGAMON for CICS on z/OS

Because the trace was active when this task ran, you can look at the combined trace for these tasks by typing T next to any of the transactions shown in Figure 10-27. The combined trace is shown in Figure 10-28, Figure 10-29 on page 179, and Figure 10-30 on page 179. The combined trace is useful to see the application logic, what program it ran, what function was invoked, what was the response, what was the offset in the program, the start time and end time for each function, and other relevant information.

The combined trace shows that GBPA (task 11992) started four asynchronous child tasks, which are highlighted in Figure 10-28. Three of these child tasks (11993, 11994, and 11995) got some storage, got content from container, and then delayed before completing. Child task 11996 got some storage, and then did some MQ calls and delays before completing.

File Edit View Tools Navigate Help 09/21/2017 08:59:47										Display : HISTORY	
Command ==> KCPTASTA CICS Task Application Trace										CICSplex : OMEGPLEX	
										Region :	
Asynchronous Trace Data for Trans GBPA Task Number 11992											
Columns 1 to 10 of 10										Rows 1 to 44 of 130	
Task Number	Type	Interval	Program	Offset	Function	Resource	Response	TCB Name	Time		
11992	TSKSTR	0.000000	GBPARENT	+0	1ST DISPATCH			QR	08:45:29.0107745332		
11992	EXECIN	0.000034	GBPARENT	+B6	GETMAIN	00000640		QR	08:45:29.0108080346		
11992	EXECOUT	0.000014	GBPARENT	+B6	GETMAIN	26D00A38	NORMAL	QR	08:45:29.0108223471		
11992	EXECIN	0.000001	GBPARENT	+144	PUT CONTAINER			QR	08:45:29.0108239721		
11992	EXECOUT	0.000019	GBPARENT	+144	PUT CONTAINER		NORMAL	QR	08:45:29.0108423471		
11992	EXECIN	0.000000	GBPARENT	+1A2	RUN TRANSID			QR	08:45:29.0108427221		
11992	EXECOUT	0.000031	GBPARENT	+1A2	RUN TRANSID		NORMAL	QR	08:45:29.0108734082		
11992	EXECIN	0.000001	GBPARENT	+200	RUN TRANSID			QR	08:45:29.0108744082		
11992	EXECOUT	0.000011	GBPARENT	+200	RUN TRANSID		NORMAL	QR	08:45:29.0108859707		
11992	EXECIN	0.000001	GBPARENT	+25E	RUN TRANSID			QR	08:45:29.0108863457		
11992	EXECOUT	0.000009	GBPARENT	+25E	RUN TRANSID		NORMAL	QR	08:45:29.0108955332		
11992	EXECIN	0.000000	GBPARENT	+2BC	RUN TRANSID			QR	08:45:29.0108957832		
11992	EXECOUT	0.000007	GBPARENT	+2BC	RUN TRANSID		NORMAL	QR	08:45:29.0109029707		
11992	EXECIN	0.000001	GBPARENT	+31A	FETCH ANY			QR	08:45:29.0109032632		
Task Switch											
11993	TSKSTR	0.000081	GBCHILD1	+0	1ST DISPATCH			QR	08:45:29.01090845419		
11993	EXECIN	0.000020	GBCHILD1	+B6	GETMAIN	00000640		QR	08:45:29.0110048544		
11993	EXECOUT	0.000008	GBCHILD1	+B6	GETMAIN	26E00A38	NORMAL	QR	08:45:29.0110124169		
11993	EXECIN	0.000001	GBCHILD1	+11C	GET CONTAINER			QR	08:45:29.0110136044		
11993	EXECOUT	0.000006	GBCHILD1	+11C	GET CONTAINER		NORMAL	QR	08:45:29.0110139169		
11993	EXECIN	0.000001	GBCHILD1	+142	DELAY	00000030		QR	08:45:29.0110205419		
Task Switch											
11994	TSKSTR	0.000053	GBCHILD2	+0	1ST DISPATCH			QR	08:45:29.0110730419		
11994	EXECIN	0.000006	GBCHILD2	+B6	GETMAIN	00000640		QR	08:45:29.0110791669		
11994	EXECOUT	0.000001	GBCHILD2	+B6	GETMAIN	26F00A38	NORMAL	QR	08:45:29.0110807919		
11994	EXECIN	0.000001	GBCHILD2	+11C	GET CONTAINER			QR	08:45:29.0110813544		
11994	EXECOUT	0.000002	GBCHILD2	+11C	GET CONTAINER		NORMAL	QR	08:45:29.0110836044		
11994	EXECIN	0.000000	GBCHILD2	+142	DELAY	00000040		QR	08:45:29.0110839169		
Task Switch											
11995	TSKSTR	0.000027	GBCHILD3	+0	1ST DISPATCH			QR	08:45:29.0111103544		
11995	EXECIN	0.000006	GBCHILD3	+B6	GETMAIN	00000640		QR	08:45:29.0111151044		
11995	EXECOUT	0.000001	GBCHILD3	+B6	GETMAIN	27000A38	NORMAL	QR	08:45:29.0111177919		
11995	EXECIN	0.000001	GBCHILD3	+11C	GET CONTAINER			QR	08:45:29.0111182919		
11995	EXECOUT	0.000002	GBCHILD3	+11C	GET CONTAINER		NORMAL	QR	08:45:29.0111206044		
11995	EXECIN	0.000000	GBCHILD3	+142	DELAY	00000050		QR	08:45:29.0111208544		
Task Switch											
11996	TSKSTR	0.000024	GBCHILD4	+0	1ST DISPATCH			QR	08:45:29.0111446044		
11996	EXECIN	0.000006	GBCHILD4	+A0	ADDRESS	COMMAREA		QR	08:45:29.0111509794		
11996	EXECOUT	0.000002	GBCHILD4	+A0	ADDRESS	FF000000	NORMAL	QR	08:45:29.0111523544		
11996	EXECIN	0.000000	GBCHILD4	+D8	GETMAIN	00024000		QR	08:45:29.0111527919		
11996	EXECOUT	0.000005	GBCHILD4	+D8	GETMAIN	27401028	NORMAL	QR	08:45:29.0111579794		
11996	RMIIIN	0.000001	GBCHILD4	+DF4	RESOURCE MANAGER	MQM		QR	08:45:29.0111586669		
11996	EXECIN	0.001052	DFHMOTRU	+1378	GETMAIN	00000060		L8006	08:45:29.0122100332		
11996	EXECOUT	0.000010	DFHMOTRU	+1378	GETMAIN	259D0F68	NORMAL	L8006	08:45:29.0122206582		
11996	RMIIOUT	0.000004	GBCHILD4	+DF4	RESOURCE MANAGER	MQM	00000000	L8006	08:45:29.0122244707		
11996	RMIIIN	0.000013	GBCHILD4	+DF4	RESOURCE MANAGER	MQM		QR	08:45:29.0122374082		
11996	RMIIOUT	0.000577	GBCHILD4	+DF4	RESOURCE MANAGER	MQM	00000000	L8006	08:45:29.0128143562		
11996	RMIIIN	0.000005	GBCHILD4	+F0C	RESOURCE MANAGER	MQM		QR	08:45:29.0128294187		

Figure 10-28 Combined trace for a group of tasks monitored by OMEGAMON for CICS on z/OS (part 1 of 3)



OMEGAMON highlighted long intervals to raise an alert, as shown in Figure 10-29.

File Edit View Tools Navigate Help 09/21/2017 08:59:59										Display : HISTORY
Command ==> KCPTASTA CICS Task Application Trace										CICSplex : OMEGPLEX
										Region :
Asynchronous Trace Data for Trans GBPA Task Number 11992										
Columns 1 to 10 of 10										Rows 45 to 90 of 130
Task Number	Type	Interval	Program	Offset	Function	Resource	Response	TCB Name	Time	
11996	RMIOUT	0.000054	GBCHIL04	+F0C	RESOURCE MANAGER	MQM	00000000	L8006	08:45:29.0128834812	
11996	EXECIN	0.000003	GBCHIL04	+1FA	SUSPEND			QR	08:45:29.0128869812	
11996	EXECOUT	0.000005	GBCHIL04	+1FA	SUSPEND		NORMAL	QR	08:45:29.0128917937	
11996	RMIIN	0.000001	GBCHIL04	+F98	RESOURCE MANAGER	MQM	00000000	QR	08:45:29.0128926062	
11996	RMIOUT	0.000062	GBCHIL04	+F98	RESOURCE MANAGER	MQM	00000000	L8006	08:45:29.0129543562	
11996	EXECIN	0.376894	GBCHIL04	+49A	DELAY	00000001	QR	08:45:29.3898485947		
11996	EXECOUT	0.106006	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:30.464345947	
11996	EXECIN	0.000005	GBCHIL04	+285	ASKTIME			QR	08:45:30.464397822	
11996	EXECOUT	0.000007	GBCHIL04	+285	ASKTIME		NORMAL	QR	08:45:30.4649469697	
11996	EXECIN	0.382233	GBCHIL04	+49A	DELAY	00000001	QR	08:45:30.8482394072		
11996	EXECOUT	0.000008	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:32.0363832905	
11996	EXECIN	0.000006	GBCHIL04	+2AC	ASKTIME			QR	08:45:32.0363894155	
11996	EXECOUT	0.000008	GBCHIL04	+2AC	ASKTIME		NORMAL	QR	08:45:32.0363972280	
11996	EXECIN	0.387249	GBCHIL04	+49A	DELAY	00000001	QR	08:45:32.4236460947		
11996	EXECOUT	0.000008	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:33.6091548447	
11996	EXECIN	0.000005	GBCHIL04	+2D2	ASKTIME			QR	08:45:33.6091597197	
11996	EXECOUT	0.000012	GBCHIL04	+2D2	ASKTIME		NORMAL	QR	08:45:33.6091719072	
11996	EXECIN	0.391109	GBCHIL04	+49A	DELAY	00000001	QR	08:45:33.9562802197		
11996	EXECOUT	0.185866	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:35.1821461047	
11996	EXECIN	0.000005	GBCHIL04	+2F8	ASKTIME			QR	08:45:35.1821519172	
11996	EXECOUT	0.000008	GBCHIL04	+2F8	ASKTIME		NORMAL	QR	08:45:35.1821579722	
11996	EXECIN	0.387037	GBCHIL04	+49A	DELAY	00000001	QR	08:45:35.5631962280		
11996	EXECOUT	0.149451	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:36.7637570422	
11996	EXECIN	0.000005	GBCHIL04	+31E	ASKTIME			QR	08:45:36.7637625422	
11996	EXECOUT	0.000007	GBCHIL04	+31E	ASKTIME		NORMAL	QR	08:45:36.7637714797	
11996	EXECIN	0.412234	GBCHIL04	+49A	DELAY	00000001	QR	08:45:37.1760051572		
11996	EXECOUT	0.151768	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:38.3277437197	
11996	EXECIN	0.000005	GBCHIL04	+344	ASKTIME			QR	08:45:38.3277489697	
11996	EXECOUT	0.000006	GBCHIL04	+344	ASKTIME		NORMAL	QR	08:45:38.3277547822	
11996	EXECIN	0.392115	GBCHIL04	+49A	DELAY	00000001	QR	08:45:38.7198694794		
11996	EXECOUT	0.106006	GBCHIL04	+49A	DELAY	00000001	NORMAL	QR	08:45:39.9006323562	
11996	EXECIN	0.000007	GBCHIL04	+36A	ASKTIME			QR	08:45:39.9006390437	
11996	EXECOUT	0.000009	GBCHIL04	+36A	ASKTIME		NORMAL	QR	08:45:39.9006435562	
11996	RMIIN	0.000001	GBCHIL04	+10B0	RESOURCE MANAGER	MQM	00000000	QR	08:45:39.9006493562	
11996	RMIOUT	0.000000	GBCHIL04	+10B0	RESOURCE MANAGER	MQM	00000000	L8006	08:45:39.900939812	
11996	RMIIN	0.000023	GBCHIL04	+113C	RESOURCE MANAGER	MQM	00000000	QR	08:45:39.9009624096	
11996	RMIOUT	0.000151	GBCHIL04	+113C	RESOURCE MANAGER	MQM	00000000	L8006	08:45:39.9011134697	
11996	RMIIN	0.000001	GBCHIL04	+E80	RESOURCE MANAGER	MQM	00000000	QR	08:45:39.9011244072	
11996	RMIOUT	0.000100	GBCHIL04	+E80	RESOURCE MANAGER	MQM	00000000	L8006	08:45:39.9012247197	
11996	EXECIN	0.000004	GBCHIL04	+464	RETURN			QR	08:45:39.9012289697	
Task Switch										
11992	EXECOUT	0.000153	GBPARENT	+31A	FETCH ANY		NORMAL	QR	08:45:39.9013917221	
11992	EXECIN	0.000002	GBPARENT	+31A	FETCH ANY			QR	08:45:39.9013933471	
Task Switch										
11993	EXECOUT	0.376894	GBCHIL01	+142	DELAY	00000030	NORMAL	QR	08:45:59.0374127197	
11993	EXECIN	0.000005	GBCHIL01	+18C	PUT CONTAINER			QR	08:45:59.0374175947	
11993	EXECOUT	0.000025	GBCHIL01	+18C	PUT CONTAINER		NORMAL	QR	08:45:59.0374424697	
11993	EXECIN	0.000001	GBCHIL01	+1DE	RETURN			QR	08:45:59.0374430322	

Figure 10-29 Combined trace for a group of tasks monitored by OMEGAMON for CICS on z/OS (part 2 of 3)

File Edit View Tools Navigate Help 09/21/2017 09:00:11									
Command ==> KCPTASTA CICS Task Application Trace									
Asynchronous Trace Data for Trans GBPA Task Number 11992									
Columns 1 to 10 of 10 Rows 88 to 130 of 130									
Task Number	Type	Interval	Program	Offset	Function	Resource	Response	TCB Name	Time
11993	EXECIN	0.000005	GBCHIL01	+18C	PUT CONTAINER			QR	08:45:59.0374175947
11993	EXECOUT	0.000025	GBCHIL01	+18C	PUT CONTAINER		NORMAL	QR	08:45:59.0374424697
11993	EXECIN	0.000001	GBCHIL01	+1DE	RETURN			QR	08:45:59.0374430322
Task Switch									
11992	EXECOUT	0.000147	GBPARENT	+31A	FETCH ANY		NORMAL	QR	08:45:59.0375904697
11992	EXECIN	0.000001	GBPARENT	+3B4	GET CONTAINER			QR	08:45:59.0375917822
11992	EXECOUT	0.000006	GBPARENT	+3B4	GET CONTAINER		NORMAL	QR	08:45:59.0375977822
11992	EXECIN	0.000001	GBPARENT	+31A	FETCH ANY			QR	08:45:59.0375982822
Task Switch									
11994	EXECOUT	0.000000	GBCHIL02	+142	DELAY	00000040	NORMAL	QR	08:46:09.2607228693
11994	EXECIN	0.000008	GBCHIL02	+1AA	PUT CONTAINER			QR	08:46:09.2607308068
11994	EXECOUT	0.000035	GBCHIL02	+1AA	PUT CONTAINER		NORMAL	QR	08:46:09.2607651818
11994	EXECIN	0.000000	GBCHIL02	+1CC	RETURN			QR	08:46:09.2607658068
Task Switch									
11992	EXECOUT	0.000023	GBPARENT	+31A	FETCH ANY		NORMAL	QR	08:46:09.2609897443
11992	EXECIN	0.000002	GBPARENT	+436	GET CONTAINER			QR	08:46:09.2609903693
11992	EXECOUT	0.000008	GBPARENT	+436	GET CONTAINER		NORMAL	QR	08:46:09.2609981193
11992	EXECIN	0.000000	GBPARENT	+31A	FETCH ANY			QR	08:46:09.2609987443
Task Switch									
11995	EXECOUT	0.351247	GBCHIL03	+142	DELAY	00000050	NORMAL	QR	08:46:19.2222150437
11995	EXECIN	0.000004	GBCHIL03	+1AA	PUT CONTAINER			QR	08:46:19.2222191687
11995	EXECOUT	0.000022	GBCHIL03	+1AA	PUT CONTAINER		NORMAL	QR	08:46:19.2222414187
11995	EXECIN	0.000001	GBCHIL03	+1CC	RETURN			QR	08:46:19.2222422937
Task Switch									
11992	EXECOUT	0.000145	GBPARENT	+31A	FETCH ANY		NORMAL	QR	08:46:19.2223973562
11992	EXECIN	0.000001	GBPARENT	+4A6	GET CONTAINER			QR	08:46:19.2223985437
11992	EXECOUT	0.000006	GBPARENT	+4A6	GET CONTAINER		NORMAL	QR	08:46:19.2223994537
11992	EXECIN	0.000001	GBPARENT	+31A	FETCH ANY			QR	08:46:19.2223994187
11992	EXECOUT	0.000000	GBPARENT	+4EA	WRITEQ TD	CSSL	NOTFND	QR	08:46:19.2223970437
11992	EXECOUT	0.000018	GBPARENT	+4EA	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2223976687
11992	EXECIN	0.000000	GBPARENT	+51C	WRITEQ TD	CSSL		QR	08:46:19.2224150437
11992	EXECOUT	0.000002	GBPARENT	+51C	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224153562
11992	EXECIN	0.000000	GBPARENT	+54E	WRITEQ TD	CSSL		QR	08:46:19.2224174187
11992	EXECOUT	0.000002	GBPARENT	+54E	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224177312
11992	EXECIN	0.000001	GBPARENT	+580	WRITEQ TD	CSSL		QR	08:46:19.2224198562
11992	EXECOUT	0.000001	GBPARENT	+580	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224201062
11992	EXECIN	0.000001	GBPARENT	+5B2	WRITEQ TD	CSSL		QR	08:46:19.2224219187
11992	EXECOUT	0.000001	GBPARENT	+5B2	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224239187
11992	EXECIN	0.000001	GBPARENT	+5E4	WRITEQ TD	CSSL		QR	08:46:19.2224241687
11992	EXECOUT	0.000001	GBPARENT	+5E4	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224259562
11992	EXECIN	0.000001	GBPARENT	+616	WRITEQ TD	CSSL		QR	08:46:19.2224261687
11992	EXECOUT	0.000001	GBPARENT	+616	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224277937
11992	EXECIN	0.000001	GBPARENT	+648	WRITEQ TD	CSSL		QR	08:46:19.2224280437
11992	EXECOUT	0.000001	GBPARENT	+648	WRITEQ TD	CSSL	NORMAL	QR	08:46:19.2224297937
11992	EXECIN	0.000001	GBPARENT	+678	SEND TEXT	GBPARE..		QR	08:46:19.2224303562
11992	EXECOUT	0.000037	GBPARENT	+678	SEND TEXT	GBPARE..	NORMAL	QR	08:46:19.2224673562
11992	EXECIN	0.000000	GBPARENT	+69A	RETURN			QR	08:46:19.2224677312

Figure 10-30 Combined trace for a group of tasks monitored by OMEGAMON for CICS on z/OS (part 3 of 3)

Based on this combined information, you can diagnose the issue and take any necessary actions, for example kill a running task from OMEGAMON.











SG24-8411-00

ISBN 0738442925

Printed in U.S.A.

Get connected

