Redbooks
ibm.com/redbooks

# Walmart and the CICS Asynchronous API: An Adoption Experience


Pradeep Gohil

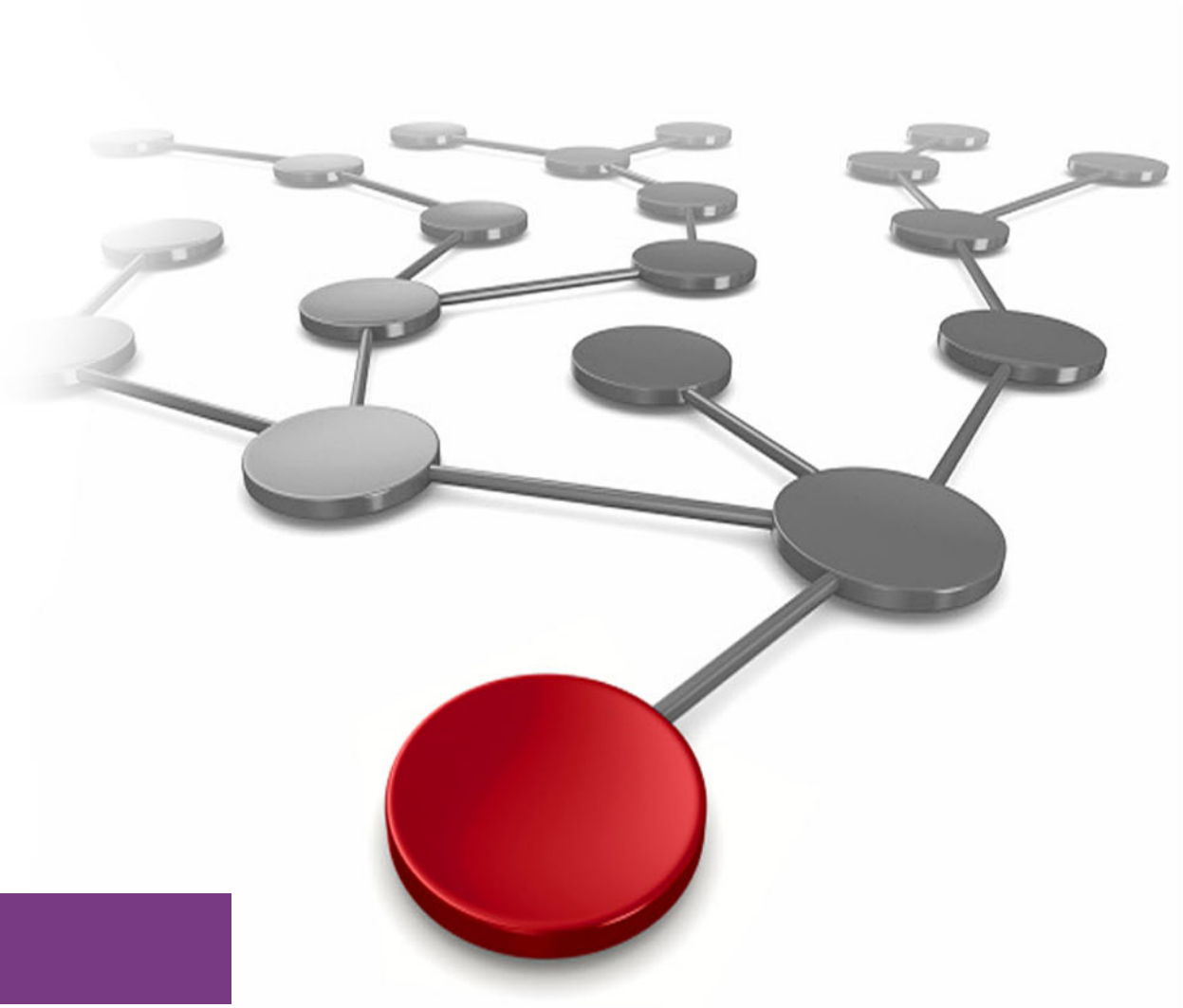
Frank De Gilio

Nick Garrod

Randy Frerking

Rich Jackson

Kellie Mathis

IBM Z

IBM

Redbooks

IBM Redbooks

# Walmart and the CICS Asynchronous API: An Adoption Experience

March 2019

SG24-8444-00

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

**First Edition (March 2019)**

This edition applies to Version 5, Release 5 of IBM CICS Transaction Server.

# Contents

**Notices** . . . . . v
Trademarks . . . . . vi

**Preface** . . . . . vii
Authors . . . . . vii
Now you can become a published author, too! . . . . . ix
Comments welcome . . . . . ix
Stay connected to IBM Redbooks . . . . . ix

**Chapter 1. Executive summary** . . . . . 1
1.1 Why Walmart chose the IBM CICS Asynchronous API . . . . . 2
1.1.1 Ease of Use . . . . . 2
1.1.2 Risk Free . . . . . 2
1.1.3 Successful Service Delivery . . . . . 3
1.2 Moving Forward . . . . . 3

**Chapter 2. Background** . . . . . 5
2.1 Asynchronous Processing . . . . . 6
2.2 IBM CICS and the CICS Asynchronous API . . . . . 7
2.2.1 IBM CICS Transaction Server . . . . . 7
2.2.2 CICS Asynchronous API . . . . . 8
2.3 Walmart and the Event Processing System . . . . . 10
2.3.1 Walmart at a Glance . . . . . 10
2.3.2 Walmart Event Processing System (EPS) . . . . . 10
2.4 Background Summary . . . . . 11

**Chapter 3. Requirements and challenges** . . . . . 13
3.1 Volume . . . . . 13
3.2 Searching . . . . . 13
3.3 Service Level Agreement . . . . . 14
3.4 Data Repository . . . . . 14
3.5 Data Structure . . . . . 14
3.5.1 Event object . . . . . 14
3.5.2 Format conversion . . . . . 15
3.6 Fundamental I/O Requirements . . . . . 16

**Chapter 4. Our initial sequential approach** . . . . . 19
4.1 Additional background and basic approach . . . . . 19
4.1.1 Search Criteria . . . . . 19
4.1.2 Sequential Processing . . . . . 20
4.1.3 Results . . . . . 21
4.2 Native VSAM . . . . . 21
4.2.1 Results . . . . . 22
4.2.2 Tradeoffs . . . . . 22
4.2.3 Disclaimers . . . . . 22
4.3 Summary . . . . . 22

**Chapter 5. Homegrown asynchronous solution** . . . . . 23
5.1 Parallel processing . . . . . 23

5.2 Asynchronous processing . . . 24
5.3 Design . . . 25
5.3.1 Prepare Data for Child . . . 26
5.3.2 Initiate Child . . . 28
5.3.3 Check for Completion . . . 31
5.3.4 Retrieve Data from Child . . . 33
5.3.5 Perform Housekeeping . . . 34
5.4 Summary . . . 36

**Chapter 6. IBM CICS asynchronous solution** . . . 37
6.1 Design . . . 37
6.1.1 Prepare Data for Child . . . 38
6.1.2 Initiate Child . . . 40
6.1.3 Check for Completion . . . 42
6.1.4 Retrieve Data from Child . . . 44
6.1.5 Perform Housekeeping . . . 45
6.2 Migration . . . 46
6.3 Summary . . . 46

**Chapter 7. Other implementation patterns** . . . 49
7.1 Unordered Responses . . . 49
7.2 Updating remote hosts (High Latency) . . . 52
7.3 Fire-and-Forget . . . 55
7.4 Summary . . . 59

**Chapter 8. Considerations** . . . 61
8.1 Transactionality and recovery . . . 61
8.2 Data integrity . . . 62
8.3 Timeouts . . . 62
8.4 CPU / Cost . . . 62
8.4.1 Command overhead . . . 63
8.4.2 Justification for additional processing . . . 63
8.5 Resources . . . 64
8.5.1 Threadsafe considerations . . . 64
8.5.2 Managing MXT and TRANCLASS . . . 64
8.6 Testing . . . 64
8.6.1 Testing that uses a single REST client . . . 65
8.6.2 Testing that uses multiple REST clients . . . 65
8.7 Skills . . . 66
8.8 Conclusion . . . 66

# 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

# Preface

This IBM® Redbooks® publication discusses practical uses of the IBM CICS asynchronous API capability. It describes the methodology, design and thought process used by a large client, Walmart, and the considerations of the choices made. The Redbooks publication provides real life examples and application patterns that benefit from the performance and scalability offered by the new API.

The book discusses the homegrown methodology used by Walmart before the API was available and compares it with the design using the new API. A discussion of the process used to migrate older applications to begin using the new API is included so the reader will understand the ease of implementing the new API. A description of real world usage patterns describes the current production application Walmart has deployed as well as other patterns to give the reader a sense of what's possible applying creative thinking with technology improvements. Finally, a section is included on the areas to be considered as you begin to plan and implement asynchronous API capabilities.

This book should be read by:

- Enterprise Architects searching for faster ways to service strategic applications across the enterprise.
- Solution Architects who want to better understand implementation possibilities for improved response times and better performance for CICS applications.
- CICS programmers looking to modernize and provide improved response times.

This book is meant to be used in tandem with IBM Redbooks publication *IBM CICS Asynchronous API: Concurrent Processing Made Simple*, SG24-8411, which will provide the background and implementation instructions and commands for the API itself.

## Authors

This book was produced by a team of specialists from around the world working at IBM Redbooks, Hursley Center.

**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 17 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 and has authored an IBM Redbooks publication on the subject (*IBM CICS Asynchronous API: Concurrent Processing Made Simple*, SG24-8411).

**Frank De Gilio** is a Distinguished Engineer from IBM. He is the Chief Technology Officer for mainframe modernization. Frank has experience in mainframe development where he focused on uniting cross platform technologies on MVS, OS/2, Windows and UNIX platforms. Frank built the IBM Design Center to help clients architect and design optimized IT infrastructures. Today Frank focuses on helping businesses modernize their mainframe environment to become critical enterprise components of their digital solutions. He has authored works on J2EE security, cloud computing, infrastructure design and management.

Frank holds a number of patents in the system software, Internet, security and infrastructure management fields.

**Nick Garrod** works in Market Enablement for IBM Z and has spent many years supporting CICS in a variety of business and marketing roles. He has been a member of the GSE Nordics steering committee for more than 15 years. Nick has been a contributor to IBM Redbooks, IBM Systems Journal, Enterprise Tech journal and Destination z. In his spare time he enjoys listening and playing music as well as traveling. Nick has been with IBM for 40 years.

**Randy Frerking** is a Distinguished Systems Engineer on the z/OS Cloud Platform team at Walmart. He is responsible for designing and developing foundational services to satisfy various IT and business requirements, providing socialization of technological concepts, influencing direction, and promoting innovation throughout Walmart's IT organization. His primary focus is on z/OS cloud and enterprise software development. Randy has more than 39 years of CICS and mainframe middleware experience with 11 years at Walmart. Randy is an IBM Redbooks Gold Author and an IBM Champion for 2019.

**Rich Jackson** is a Principal Systems Engineer at Walmart where he is responsible for the design, development, and operation of foundational, managed services that enable application developers to satisfy an array of business needs. He regularly speaks on topics such as cloud computing, services, and DevOps as they relate to enterprise mainframe environments. Rich's background includes storage and z/OS management, grid computing planning and design, HA/DR implementations, and service delivery. His experience with large-scale, global IT operations provides a perspective that is highly valued among his peers. Rich is an IBM Redbooks Gold Author and an IBM Champion for 2019.

**Kellie Mathis** is an IT executive who has worked in the area of mainframe ecosystem for 34 years. She retired from IBM in 2017 and now leads the mainframe team at Direct Systems Support. Kellie's background includes z/OS, Z systems hardware, Z software, storage and HA/DR. She regularly consults with clients to help them get the best value from their mainframe environment, focusing on cost structures, organizational issues, and exploitation of modern mainframe practices. Kellie is an IBM Redbooks Gold Author.

*Figure 1 Front: Nick, Kellie, Randy; Middle: Frank, Pradeep; Top: Rich*

The project that produced this publication was managed by **Martin Keen**, IBM Redbooks Project Leader and edited by **Matthew Lesher**.

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

- Greg Lubel
- Gerry McFadden
- Tannia Sanchez
- Joseph Sayer
- Chris Poole
- Jenny He
- Samantha Catling
- Ashleigh Denholm
- Russ Ian Winter

## 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 IBM Redbooks 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, IBM Redbooks
  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

1

# Executive summary

IBM Customer Information Control System (CICS®) has long been an industry standard for rapid, high-volume online transaction processing. Now, IBM has developed the CICS asynchronous API interface, which dramatically improves response times for applications that are compatible with asynchronous programming techniques. This new capability helps application developers service requests faster and provides the correlation that they need to manage an asynchronous model. Developers can use asynchronous programs in all CICS-supported languages, so teams benefit from flexibility and reuse of skills.

Consider the microwave oven, a tool now used in most homes. It was invented by Percy Spencer as he experimented with a new vacuum tube called a magnetron. He was surprised when a candy bar in his pocket melted, so he did another experiment with popcorn. When it started popping, he realized he was onto something. In 1947, Spencer built the first microwave oven while working for Raytheon. It was called the Radarange, and at 5 1/2 feet tall and weighing 750 pounds, it cost $5000. It was usable but so large and difficult to control that it was hard to imagine the technology could become a standard tool. However, 20 years later, in 1967, a much smaller version incorporated the magnetron and controls into a countertop package, with an interface that was easy for the average consumer. And at a cost of only $495, it was a winner.

The story of asynchronous processing is similar to that of the microwave. It was possible for a long time, but challenging to apply enterprise grade applications, which have these challenges:

- Asynchronous calling patterns
- Task queues that are difficult to coordinate the many tasks
- Sharing data among tasks
- Maintaining synchronization of work

Yet, similar to the modern microwave, the CICS asynchronous API now represents a standard tool that application developers use to master asynchronous calling patterns in high-volume transaction processing.

Walmart has successfully deployed applications that use the IBM CICS asynchronous API and shares their experiences in this book. The details of this pilot application methodology prove what can be achieved on a large scale. And this methodology rests on a foundation of the underlying qualities of service that is expected for IBM Z® applications. Walmart runs an

application that does event processing across their worldwide enterprise. An event is any number of business things that happen across any line of business in the organization. An example would be a truck that is completing a delivery of goods. The event is represented by the metadata about that point in time. With the size of Walmart's business, there are massive amounts of events. The business problem is finding a way to search an interval that consists of 5 million events to find 'the needle in the haystack' whilst meeting the service level agreement.

Walmart tried to use standard CICS APIs and could achieve only 5000 I/Os a second, a rate that was well below the objective of 1 million I/Os a second. The next effort was to try splitting the workload and use native calls. Those results exceeded 60,000 I/Os a second, but clearly were still short of the goal. Walmart then created a homegrown asynchronous method. Although it provided better response times, it was hard to maintain.

After the IBM CICS asynchronous API became available, Walmart switched from the homegrown asynchronous method to the IBM CICS asynchronous API. The reduction in response time went from 2 minutes down to 1 second. The result is this:
**By using IBM CICS asynchronous API, Walmart enhanced a complex search capability to achieve large scale transactions in minimal time.**

## 1.1 Why Walmart chose the IBM CICS Asynchronous API

The potential benefits of using the asynchronous API are explained below. Historically, CICS applications ran single-purpose synchronous application patterns. Applications needed to fetch data from across all parts of an organization. The response time could suffer while waiting for the results, and service level agreements could be missed. Manual ways exist to achieve CICS asynchronous functionality, but they require manual management and are complex. Deploying the asynchronous API provides concurrency in a simplified programming model that can be used across platforms.

### 1.1.1 Ease of Use

Walmart's application needed to search through 5 million events with an SLA of 2 seconds. It was apparent to Walmart that to process that much data, they would need to distribute the search processing then aggregate the responses. Walmart found their homegrown asynchronous process required manual setup of temporary storage queues, shared storage, and abend logic. The IBM CICS asynchronous API takes care of the monitoring function for you, including child-state information. When the processing is complete, CICS takes care of the cleanup, deleting channels and containers, releasing memory, and so on.

### 1.1.2 Risk Free

IBM CICS asynchronous API mitigates risk as follows:

- **Timing and Correlation** is managed so as to avoid timing window errors. CICS correlates data seamlessly for the developer and helps avoid common errors.
- **Resource management** is handled by CICS to be sure that storage is freed up in the right timing window, clean up happens on time to avoid stale data, and child tasks are terminated.
- **Task management** between parent task and child task is handled automatically, which allows asynchronous replies to be consumed in a timely manner. In addition, orphaned child tasks can be managed effectively.

### 1.1.3 Successful Service Delivery

IBM CICS asynchronous API brought these benefits:

- Walmart increased the number of requests in an application and still met SLAs with improved response time.
- Developer skills already existed to deploy application extensions.
- The environment is easier to manage in comparison to other asynchronous solutions.
- The asynchronous API commands are threadsafe, so they save CPU time and response time.

> "This is really a cool service … we are using the functionality CICS has provided in a new and creative way." Randy Frerking, Walmart

## 1.2 Moving Forward

Subsequent chapters cover these topics:

- Chapter 2: Background information on asynchronous processing, the CICS asynchronous API, and Walmart's pilot application.
- Chapter 3: Discussion of the new business requirements, possible solutions, and SLA that Walmart faced.
- Chapter 4: Discussion of Walmart's initial sequential solution that was based on traditional methods.
- Chapter 5: Description of Walmart's homegrown asynchronous solution and design.
- Chapter 6: Details of Walmart's new design that uses the CICS asynchronous API and migration considerations.
- Chapter 7: Other implementation patterns for the reader to consider.
- Chapter 8: Considerations for planning deployment of asynchronous solutions.

2

# Background

In this Redbooks publication, we follow the experience of the Walmart delivery team as they embrace asynchronous processing patterns in enterprise-grade applications.

We follow their stepped approach, starting with the initial problem space and how they assessed an ambitious set of requirements. In the end, we see how they arrived at a high-throughput, scalable service that minimizes application response times.

We see the benefits that they realized from adopting an asynchronous processing pattern. And we also share the key choices and challenges that they faced.

As with all user experience stories, the benefits that you can realize depend heavily on application architectures and existing assets that you start with. All figures that are quoted are measurements that the Walmart team took. Your results might vary. However, this fact is certain: the Walmart service transformed a 2-minute response time to just 1 second!

There are two main parts to this background chapter:

1. We introduce IBM CICS Transaction Server (CICS TS) as a multi-language application server. We explain how this platform provides an asynchronous application programming interface where you can easily develop and manage asynchronous programming patterns.
2. The Event Processing System (EPS) that Walmart developed. EPS is a large data application that provides a central point for events that are produced and consumed across the Walmart organization.

The chapters that follow this background chapter describe new requirements of the Walmart application and follow the developers' iterative solutions, which include these milestones:

- The initial "traditional-style" sequential programming implementation.
- An implementation that minimized response time, based on a home-grown asynchronous framework.
- Adoption of the IBM CICS asynchronous API capability to arrive at an implementation that minimizes response time and reduces risk and maintenance costs.

The final chapters look at other asynchronous patterns that Walmart is developing and also highlight key choices when you work with such techniques.

## 2.1 Asynchronous Processing

The paradigm of asynchronous processing is not new in the world of computing, nor to human behavior. Consider these contrastive examples:

- **Sequential serving model:** Customers wait in a queue to buy popcorn from a single cashier at the cinema. When a customer gets to the head of the queue, and they are served. They make their order, wait for the popcorn to be served, pay for it, and complete their transaction. Then, they leave the queue, and it is the turn of the next customer in line.
- **Asynchronous model:** A restaurant setting contrasts with the popcorn queue. Multiple groups of diners are seated at tables whenever they arrive. Each table of diners progresses at their own pace. Coordination is done by the restaurant. The restaurant processes tables to reduce response time and to maximize throughput and satisfaction. Table 2 can order and eat their food, request their bill, and complete their transaction, without the need for Table 1 to complete first.

The restaurant example reveals the crux of asynchronous processing. If you can serve/request a service, without being blocked by it, then you are able to follow other paths. This might mean continuing the algorithm, processing more data, calling another service concurrently, or even … serving another table in the restaurant.

If you can work on different parts of a service concurrently, even though those constituent parts take a set amount of effort, the overall service response time can be reduced. Basically, you make better use of your time!

Over the 50 years of CICS, application developers have implemented their own asynchronous solutions for CICS applications. Often, these frameworks involve piecing together technologies, including these:

- **`EXEC CICS START`** API
- CICS Business Transaction Services
- CICS Event processing
- Passing data by using TD queues / TS queues / Shared **`GETMAIN`**s and **`FREEMAIN`**s
- Returning control by using set times / polling code by using the **`DELAY`** API / posting ECBs
- Using other products, such as IBM MQ
- And other IBM and non-IBM technologies

This list shows that there is no single method to implement asynchronous patterns in CICS. Also, these technologies are often used for scenarios other than what they were originally designed for. For example, the **`START API`** might be used with proprietary **`GETMAIN`**s/**`FREEMAIN`**s to pass data, thereby gaining a response on a Fire-and-Forget API. And this mechanism might be coupled with suboptimal polling algorithms that use looping **`DELAY`** calls to check for response arrivals.

Research also reveals how asynchronous programming techniques tend to be alien to traditional CICS developers. Many asynchronous projects have been abandoned or had limited success, due to factors like these:

- Lack of skills
- A brittle offering with a high rate of failure
- High levels of complexity
- Higher cost of development
- Increased cost of on-going maintenance

A common difficulty is the exposure of difficult-to-test timing windows. For example, a service might be slow to respond, does not respond at all, or returns multiple responses for multiple

instances. As a result, data becomes out of sync. This scenario has caused some high-profile cases where the wrong personal data was served to incorrect recipients in production.

Rare failures might be seen as acceptable in some environments. However, in enterprise-grade applications, such as those served by CICS, the high throughput and transaction rates that are required can easily expose problems that are related to timing windows.

The CICS asynchronous API is designed to be a simple to use, yet powerful, IBM-supported set of APIs that support the implementation of asynchronous processing patterns in CICS applications.

## 2.2 IBM CICS and the CICS Asynchronous API

In this section, we take a closer look at the IBM CICS technology that helps you adopt the high transaction rate and simplified asynchronous pattern.

### 2.2.1 IBM CICS Transaction Server

IBM introduced modular mainframe computers (System 360) in the mid 1960's and shipped rudimentary software at no extra charge. In 1969, IBM introduced more sophisticated software to run on their mainframes, and one of these products was called Customer Information Control System (CICS). CICS originated with the need in the public utility industry to update their repositories online from multiple sources, without locking out those repositories. Very quickly it became clear that this software was destined for a much wider community than just the public utilities.

CICS was intended to be in-the-middle software (Middleware) to manage processes across the operating system, data repositories, application and business logic, and to coordinate communication between these elements. In areas where it made sense, CICS would take management responsibilities, as in these examples:

- Originally CICS kept its own logs.
- The need for subdispatching work on a single TCB meant that CICS architecture required its own dispatcher.

Since this first release, CICS has continued to evolve. It adopts new technologies where appropriate, and it takes advantage of enhancements that are made in the operating system. Also, CICS maintains its reputation as the premier mixed-language application server. When you couple the qualities of service of running CICS with IBM Z hardware, you get a premier mixed-language application server that achieves these enterprise-grade features:

- Excellent performance
- Unimpeachable security for integrity of information
- High resilience with the ability to scale to a massive extent

To make it easy for programmers to write applications to run in CICS, the CICS Transaction Server V5.4 provides help with asynchronous processing in the form of the CICS Asynchronous API. Walmart use of this API in IBM Z with CICS resulted in throughput that was beyond belief, as compared to the existing processes.

> "For over a decade, prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers." IBM Fellow Dr. Gene Amdahl in 1967.
>
> (Amdahl's research alerted computer scientists to the complexity and limitations of parallel computing.)

### 2.2.2 CICS Asynchronous API

The CICS asynchronous API is designed to be a simple, yet powerful API that application developers can use to confidently deliver asynchronous patterns into CICS applications.

The CICS asynchronous API was added to CICS Transaction Server V5.4, and the full breadth of the capability includes these features:

- A set of threadsafe EXEC CICS and JCICS API commands.
- Monitoring enhancements.
- Statistics enhancements.
- Transaction tracking enhancements.
- Trace and dump enhancements.
- New CICS policy threshold type.
- Tooling support.
- The API, which covers the three main requirements of asynchronous processing:
  a. Initiation of an asynchronous transaction to execute supporting work, without blocking the initiator.
  b. Efficient return of results from the work that is initiated.
  c. Safe passing of data.

The parent-child relationship is at the heart of the CICS capability. A parent can have zero to many child associations. All parent and child entities are themselves CICS transactions. This approach makes their behavior predictable and easy for experienced CICS application developers to understand. The difference between these and 'regular' CICS transactions is that CICS maintains an internal understanding of scopes and relationships. That way, the CICS system can pass data through CICS channels, and also safely manage task-termination activities. Many of the hurdles that face the application developer are thereby removed.

For example, a child transaction can be initiated and it might complete and terminate before the parent is ready to fetch the results. This is not a problem for CICS. CICS maintains the child's channel data until the parent is ready to request the channel. In other situations, the parent might decide to not fetch the results of numerous child transactions. Instead it simply terminates its execution. In this situation, the CICS system is aware that the parent has terminated, and cleans up remaining channels associated with child tasks.

In addition to scenarios that involve numerous child transactions, the CICS asynchronous API also provides benefits in single-child scenarios. For example, when the parent calls an external service via HTTP, the parent is typically blocked until the external service responds. By running a child transaction to invoke the web service, the parent is not blocked by the service call. The parent can continue with other work. Perhaps the parent can take advantage of the timeout feature of the CICS asynchronous API and provide a responsive service to its initiator.

The CICS Asynchronous API includes the following set of threadsafe EXEC CICS commands:

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

## RUN TRANSID

A CICS transaction can use the **RUN TRANSID** command to begin another child transaction. With this command, you can specify a Channel name to pass container data to the child transaction. The command returns a child token parameter that is used to reference the child transaction in other CICS asynchronous API commands.

## FETCH CHILD

The **FETCH CHILD** command enables a parent transaction to retrieve data from a completed child transaction. The child token that is issued on the **RUN TRANSID** command identifies which child transaction details to retrieve. Data in containers can be returned to the parent transaction through a new Channel that is returned when the API command executes.

By default, the **FETCH CHILD** command blocks until the child transaction has terminated. An optional TIMEOUT parameter allows the issuing parent transaction to specify the length of time that it is willing to wait for the child transaction to complete.

## FETCH ANY

The **FETCH ANY** command works in a similar fashion to the **FETCH CHILD** command. The main difference is that the **FETCH CHILD** command specifies the child transaction that it is interested in. In contrast, the **FETCH ANY** command returns the results of any one of the child transactions that have completed (that have not been fetched yet).

A return parameter on the **FETCH ANY** command specifies the child token of the details of the returning transactions. This can be matched against the **RUN TRANSID** child parameter to identify which child transaction is being returned.

You can issue multiple **RUN TRANSID** commands, and then issue multiple **FETCH ANY** commands to retrieve the results from those child transactions. The parent transaction can then process the child transactions as soon as they complete, in any order. This way, the response time of the application is minimized.

By default, the **FETCH ANY** command blocks until at least one child transaction has completed (or there are no further unfetched child transactions). An optional **TIMEOUT** parameter allows the issuing parent transaction to specify the length of time that it is willing to wait for a child transaction to complete.

## FREE CHILD

The **FREE CHILD** command is issued by a parent transaction to disassociate the child transaction that is specified.

When a child transaction is freed, internal control blocks are removed and, if it exists, the child's unfetched channel data is not preserved after the child terminates. Issuing the **FREE CHILD** command does not prevent, nor halt the child transaction from executing.

In short-lived applications, it might not be necessary to issue **FREE CHILD** commands, because control blocks and channels are cleaned up by CICS when the parent and child

transactions are completed. However, in some situations you might benefit from issuing the **FREE CHILD** command, in applications that include these conditions:

- Long running parent transactions
- Large data in channels
- High number of child transactions
- Unrequired information that is returned from child transactions

In this background section, we shared an overview of the CICS asynchronous API capabilities and looked at the individual API commands. For more details, including step-by-step examples, read Redbooks publication *IBM CICS Asynchronous API: Concurrent Processing Made Simple*, https://www.redbooks.ibm.com/abstracts/sg248411.html.

## 2.3 Walmart and the Event Processing System

In this section, we take a closer look at the Walmart environment and their Event Processing System (EPS). This Walmart-developed service forms the use case for most of this Redbooks publication.

### 2.3.1 Walmart at a Glance

Walmart is a large user of IBM's CICS technology. Walmart deployed CICS in a parallel sysplex environment to gain the ability to scale and preserve availability that is designed to keep the systems running. Walmart runs approximately 750 million CICS transactions a day that are deployed across multiple environments and data centers.

Walmart regularly shares their experiences and encourages others to do the same. They hope that shared learning will lead to more creative technological advances. Look for other Walmart experiences in IBM Redbooks publications on similar topics including these:

- *Creating IBM z/OS Cloud Services*, SG24-8324, https://www.redbooks.ibm.com/abstracts/sg248324.html
- *How Walmart Became a Cloud Services Provider with IBM CICS*, SG24-8347, https://www.redbooks.ibm.com/abstracts/sg248347.html
- *IBM CICS and the Coupling Facility: Beyond the Basics*, SG24-8420, https://www.redbooks.ibm.com/abstracts/sg248420.html

In addition, Walmart regularly contributes to the open source world and has contributed three IBM z/OS® cloud services that use CICS. These can be found at the following GitHub repositories. Walmart encourages downloads and interaction with other users.

- zUID: z/OS based unique identifier generator (https://github.com/walmartlabs/zUID)
- zECS: z/OS based enterprise cache service (https://github.com/walmartlabs/zFAM)
- zFAM: z/OS based file access manager (https://github.com/walmartlabs/zECS)

### 2.3.2 Walmart Event Processing System (EPS)

The Event Processing System (EPS) service, as its name suggests, is centered around the production and consumption of events in the Walmart organization.

Figure 2-1 on page 11 shows the overall framework for EPS. Around the outside of the architecture are the Systems of Engagement (SoE). Various business areas can sign up in

EPS to produce events. Typically, these areas automate the generation of events for EPS to record. For example, delivery shipments produce an event when a truck trips a sensor on arrival at the depot, when the POD is signed as being received at the final destination, when the vehicle stops to refuel, and so on. These SoEs are known as the event producers.

The events are initially processed by a highly distributed cloud application. Then, they are sent to be stored centrally in the Systems of Record (SoR), as an entry in a file for VSAM/RLS (Virtual Storage Access Method with Record Level Sharing) on IBM Z. The qualities of service on mainframes enables the storage of the large amount of data that the event producers generate. The EPS service is projected to publish 500 million events per day. In addition, between 7 to 30 days of events are stored.

Business areas, in addition to registering as event producers, can also register as event consumers. Typically, analysis and consumption of the events are human-led activities.

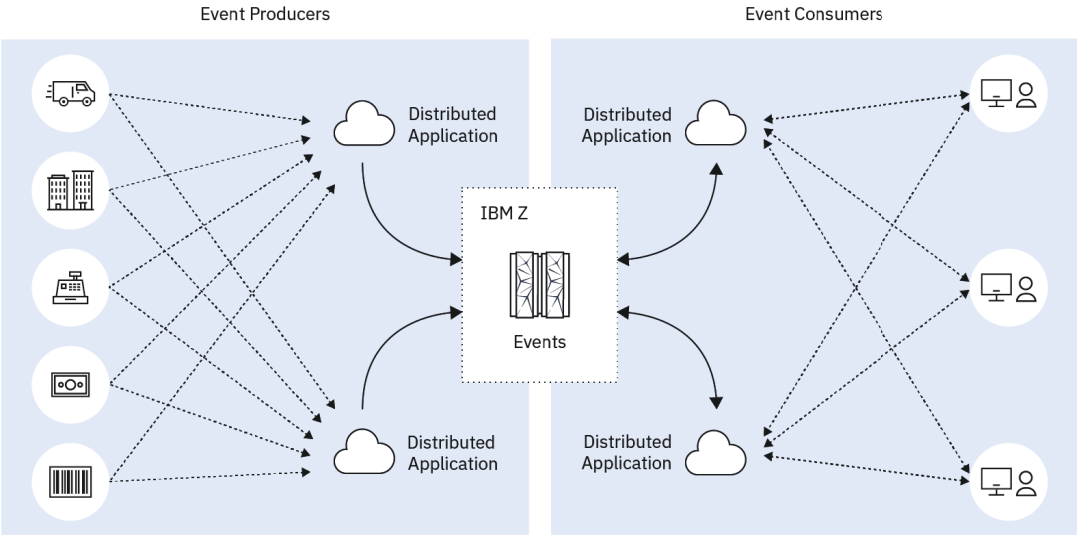



*Figure 2-1 Event Processing System*

Each event that is logged in EPS is that of a distinct incident. There are two parts to every event entry, and these parts form the entire event in the log.

*Table 2-1* Parts of an EPS event

| Part | Description |
|---|---|
| Event key | Consists of the date and time, among other details. The key serves the following purposes:<br>► Searching the events based on time.<br>► Distributing sets of events for processing. |
| Event payload | The data stored by the event publisher. Its format and size differ between different publishers. It conforms to the publisher's schema at time of first registration to EPS. |

## 2.4 Background Summary

In this chapter, we introduced the motivation for asynchronous processing and the benefits and challenges that it can bring to enterprise-grade applications. We also reviewed why

large-scale operations depend on the robustness and reliability of IBM Z and on services like these:

- The CICS Transaction Server (CICS TS, the world's premier mixed-language application server) and why it makes such a good platform to develop and serve enterprise applications.
- The capabilities of the CICS Asynchronous API that CICS TS V5.4 introduced, which greatly simplify the delivery of asynchronous application patterns.
- The Walmart Event Processing System (EPS) application, especially its data lake of events that is open to all parts of the Walmart organization to not only produce but consume events from.

In Chapter 3, “Requirements and challenges” on page 13, we look at the new requirements for EPS. A staged approach in Chapter 4, “Our initial sequential approach” on page 19 shows how the delivery team investigated a traditional sequential solution to the problem. However, that approach failed to address the aggressive targets. So, we see in Chapter 5, “Homegrown asynchronous solution” on page 23 how the team rearchitected the application processing pattern to implement a 'homegrown' asynchronous architecture. Lastly, Chapter 6, “IBM CICS asynchronous solution” on page 37 shows the benefits of adopting the CICS asynchronous API.

Having proved the success of adopting the CICS asynchronous API with their EPS application, Walmart has related projects under way. In Chapter 7, “Other implementation patterns” on page 49, we explore a subset of those projects that use a range of asynchronous processing patterns.

In Chapter 8, “Considerations” on page 61, we discuss themes and areas of interest that the Walmart team discovered while it worked with asynchronous patterns in CICS applications. It is likely that many clients that have similar projects will have similar areas of interest. So, be sure to read how Walmart approached these situations in Chapter 8, “Considerations” on page 61.

3

# Requirements and challenges

This chapter explores the challenges that are associated with the application's requirements. The following user-centric statement summarizes the high-level business requirement:

> "*As a Walmart business unit associate, I need the ability to request information about granular activities or events that are related to the operations of my business unit, and to receive that information within 2 seconds of my request*."

The following sections review the technical details of reaching this objective. Factors include data volume, structure, storage type, and input/output (I/O) rates.

## 3.1 Volume

As described in 2.3.2, “Walmart Event Processing System (EPS)” on page 10, many business areas can subscribe as publishers to the event processing framework. The overall quantity of events can be large, so the volume of data becomes a consideration. The projections for event publishing were 5 million events per day upon initial application deployment and rapidly increasing to 500 million events per day, as more publishers come online. Additionally, the application required retention periods of 7 - 30 days for the events, further increasing the overall volume.

## 3.2 Searching

In a situation like this, capturing and storing high volumes of data is typically not too concerning. However, the ability to search through large amounts of these records quickly can be challenging. The application services large volumes of events that include dynamic indexable attributes that are associated with business data and metadata. The searches against the data are also very dynamic, with criteria for event types, attributes, and search range all being highly variable. Additionally, any search results with multiple entries must be returned in chronological order. So, the search capability was immediately identified as the primary problem to be addressed.

## 3.3 Service Level Agreement

The term "quickly" is obviously a relative statement that depends on many factors. Most importantly for this application, the searches are primarily driven by human interaction with the actual user who expects near-real-time results in a client web application. So, the Service Level Agreement (SLA) for the responses is set to a maximum of 2 seconds to maintain a quality user experience. Within the context of this publication, the term "quickly" relates to this metric of less than or equal to 2 seconds.

## 3.4 Data Repository

The application team initially implemented the data repository component of this application on a distributed database product. However, they encountered many problems with stability and an inability to achieve the response-time goals. Other distributed database products were considered. But the application team instead approached the Walmart z/OS Services Engineering team (zServices team) with the challenge. That team previously demonstrated success with other zServices products and how those products operate at scale.

All zServices products that store data use centralized storage that is shared among numerous systems. The file storage that is used is Virtual Storage Access Method with Record Level Sharing (VSAM/RLS), which allows concurrent read and update accessibility from numerous processes. This unique characteristic of the z/OS platform allows broad access to data. At the same time, it avoids many of the complexities and challenges with managing distributed data. For more information about Walmart's use of VSAM/RLS, see *How Walmart Became a Cloud Services Provider with IBM CICS*, http://www.redbooks.ibm.com/abstracts/sg248347.html?Open.

## 3.5 Data Structure

This section examines the event object and what it represents. Then it reviews how the format of the event can be adjusted to accommodate processing expectations.

### 3.5.1 Event object

The events are simply collections of information that is presented in a canonical format. At present, either JavaScript Object Notation (JSON) or Extensible Markup Language (XML) formats are used. The following simplified example of an event is represented in JSON, because that format is most commonly used:

```
{
  "eventType": "US|000001721|0001|2019-02-11T08:00:35.002Z",
  "eventID": "c3BlZWRiYWxs==",
  "producerId": "2e7d0f3f00d5acdc497b358779fa1b6f",
  "statID": "0000000042"
  "locID": "13355"
}
```

This sample event includes these details:

- eventType - includes operating region, business unit, subunit, date/timestamp
- eventID - identifies specific event
- producerID - entity that generated the event

- statID - value that represents some status
- locID - identifies a specific geographic location

In computer systems, the values that are being processed are commonly cryptic and carry little meaning to a human observer. To better understand what this event might actually represent, some assumptions can be made for what each of the values in this event signifies (Table 3-1):

*Table 3-1 Event object*

| Value label | Value | Description |
|---|---|---|
| eventType | US\|000001721\|0001\|2019-02-11T08:00:35.002Z | Identifies this event as being generated by the Transportation division in the U.S. at 08:00:35 on February 11, 2019. |
| eventID | c3BlZWRiYWxs== | A specialized value that provides uniqueness to the event entry. |
| producerID | 2e7d0f3f00d5acdc497b358779fa1b6f | Unique string that identifies a particular vehicle in the fleet. |
| statID | 0000000042 | Code that represents a status of "ARRIVED". |
| locID | 13355 | Location number that equates to a Distribution Center in Derry, Maine. |

With these value associations established, the preceding event can be reproduced as the following human-readable statement:

```
"The U.S. Transportation fleet vehicle number 2e7d0f3f00d5acdc497b358779fa1b6f
arrived at the Derry, Maine Distribution Center at 08:00:35 on February 11, 2019."
```

## 3.5.2 Format conversion

The team decided at an early point to convert the canonical format of the events to a columnar structure in the data store. This change accommodates the search function as follows:

- The columns (or positional fields) represent the indexable attributes of the events, potentially along with more fields to include other relevant characteristics.
- Subgroups of these fields can then be defined as composite keys. This approach facilitates the use of initial search parameters for quick identification. Then, other attributes can be located within those entries to further filter results.

Of particular note in this scheme, each event is timestamped and all searches are based on time ranges. An "event ID" attribute is also included in the event to provide uniqueness. That way, time stamp (included in the eventType value) and event ID are included in a composite key. Other attributes can be included in the composite key based on the published events and search mechanisms that are provided, but those scenarios are not covered here.

The columnar structure and composite key are denoted in a definition list, which is used by the EPS (Event Processing System) service. In the following example, the columns or fields that are associated with each item in the JSON event are defined. Additionally, a top-level entry identifies the fields that are included in the composite key. (Notice the Len= value, which is the sum of the lengths of the eventType and eventID fields.)

```
ID=001,Col=0000001,Len=000056,Type=C,Name=eventKey
```

```
ID=001,Col=0000001,Len=000042,Type=X,Name=eventType
ID=001,Col=0000002,Len=000014,Type=C,Name=eventID
ID=001,Col=0000004,Len=000032,Type=C,Name=producerID
ID=001,Col=0000005,Len=000010,Type=C,Name=statID
ID=001,Col=0000006,Len=000020,Type=C,Name=locID
```

The conversion of the event object into this columnar format effectively "flattens" the information into an individual row of concatenated values. The primary identifying components are located at the beginning of the entry. Put another way, the event can now be viewed as a key and an associated payload, as represented in Figure 3-1.

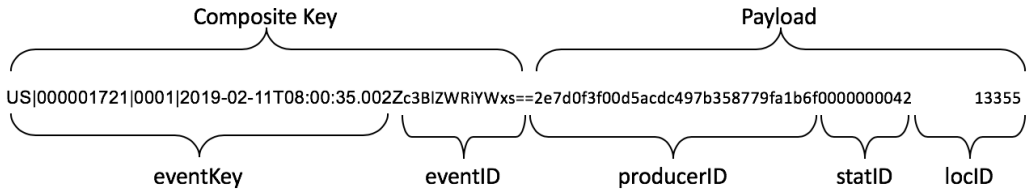



*Figure 3-1 Key and Payload*

The new format now fits nicely (as a key and value) into the Key-Sequenced Data Store (KSDS) VSAM/RLS structure that can be used for searching.

This data structure was not an explicit functional requirement for the service. On the other hand, it was a necessary technical condition for achieving the non-functional requirements of the application. This structure proved to be highly relevant in subsequent testing and design decisions.

> These event examples have been highly simplified for demonstration purposes and to maintain proprietary rights.

## 3.6 Fundamental I/O Requirements

Ultimately, the ability to satisfy the application's response-time goals depends on the ability of the EPS search service to process all relevant records within that time frame. The overall volume of data includes 500 million events per day for up to 30 days. Nonetheless, search criteria restrictions in the client interface effectively limit this processing to subsets of the data. Each search request is restricted to the following boundaries:

- A single operational region
- A single business unit
- A 24-hour period

As reviewed in 3.5, “Data Structure” on page 14, each of the preceding values is included in the key for each event in a KSDS VSAM/RLS data store. As a result, direct access to the relevant divisions of the data is possible, and the search service I/O activity is also confined to the pertinent subset of events.

The application team projected the maximum number of events for a given operational region/business-unit combination to be 5 million per day. In reality, most of these groupings were projected to generate about 2 million events per day. Additionally, it was expected that even for the largest data sets, the typical search pattern would be limited to ranges that hover around 2 million events.

This information and the stated response time SLA of 2 seconds established in 3.3, “Service Level Agreement” on page 14 establish the processing requirements of the search service. Table 3-2 illustrates the calculations of the typical and atypical processing requirements.

*Table 3-2 Processing requirements of the search service*

| **Total Events to Search** | **Response Time SLA** | **Required Processing Rate (Total Events/Response Time SLA)** |
|---|---|---|
| 2,000,000 (Typical) | 2 seconds | 1,000,000 per second |
| 5,000,000 (Outlier) | 2 seconds | 2,500,000 per second |

The "typical" scenario of 1 million events per second was deemed acceptable and became the primary requirement. The outlier scenario was reserved as a stretch goal. The following chapters examine these issues:

- Initial approaches to this challenge
- How eventually asynchronous processing with the IBM CICS asynchronous API was used to achieve these goals.

4

# Our initial sequential approach

As discussed in Chapter 3, “Requirements and challenges” on page 13, the reading at least 1 million records per second was expected to satisfy the minimum requirements of the application. This chapter describes the initial testing that benchmarked the I/O rates and determined the feasibility of the objective.

## 4.1 Additional background and basic approach

To provide the capabilities that the application required, the EPS (Event Processing System) search service followed this process:

1. Obtain search criteria from a client request.
2. Peruse the data set that is associated with the search criteria to locate matching events.

This section describes the components that were used to perform these preliminary functions.

### 4.1.1 Search Criteria

In this approach, the search criteria is sent from the client to the EPS service through the distributed cloud application that is described in 2.3.2, “Walmart Event Processing System (EPS)” on page 10. The primary means of communication for this approach is Hypertext Transfer Protocol (HTTP). A pre-existing custom format (developed by Walmart as part of an earlier service) was used to accommodate these queries. In this format, you pass a specialized SQL-like (Structured Query Language) query string on HTTP GET requests by prefixing the string with the "zQL" (zServices Query Language) keyword.

As described in 3.5, “Data Structure” on page 14, the searchable events are canonical objects (typically JSON) that are converted into a flat columnar format for storage, as you see in Figure 4-1.

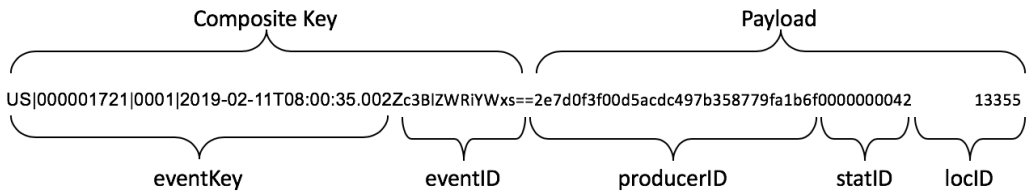



*Figure 4-1 Columnar structure*

The field or column names can be referenced in query strings in the zQL format via HTTP GET requests. The example in 3.5.1, “Event object” on page 14 describes a single event that is related to vehicle number `2e7d0f3f00d5acdc497b358779fa1b6f` (identified by the producerID value) of the U.S. Transportation fleet (derived from the eventKey value). Suppose that you needed to satisfy the following related request:

```
"Please show any events associated with U.S. Transportation fleet vehicle
2e7d0f3f00d5acdc497b358779fa1b6f that occurred between 08:30 and 10:15 on February
11, 2019"
```

This request in the zQL format would resemble this query string:

```
https://hostname.com/EPS/?zQL,SELECT,(FIELDS(eventKey),(eventID),(producerID),(sta
tID),(locID)),(WHERE(eventKey+US|000001721|0001|2019-02-11T08:30:00.000Z),(eventKe
y-US|000001721|0001|2019-02-11T10:15:00.000Z),(producerID=2e7d0f3f00d5acdc497b3587
79fa1b6f))
```

CICS gets this request through the **`WEB RECEIVE`** and **`WEB EXTRACT`** commands, then parses the request to determine the search criteria. The specific syntax of these query strings is not relevant for this document. In general, the strings confirm these facts:

- Search parameters can be passed to the EPS service as a query string.
- These strings adhere to the structure outlined in 3.5, “Data Structure” on page 14.

For more information about zQL, see `https://github.com/walmartlabs/zFAM`.

### 4.1.2 Sequential Processing

Our initial test relied on basic sequential processing. Upon receiving the search criteria, the EPS service read the data set in search of the requested events. Sections 3.5, “Data Structure” on page 14 and 3.6, “Fundamental I/O Requirements” on page 16 explained how the key in a KSDS VSAM/RLS data store gives direct access to the appropriate subset of data that must be processed. Using the search example from 4.1.1, “Search Criteria” on page 19, the following parameters (all contained within the eventKey field) can be used as an example:

- Region = US
- Business Unit = 1721 (Transportation)
- Start date/time Range = 2019-02-11T08:30:00.000
- End date/time Range = 2019-02-11T10:15:00.000

With this information, the EPS search process navigates directly to the first relevant entry in the data store, as illustrated in Figure 4-2.

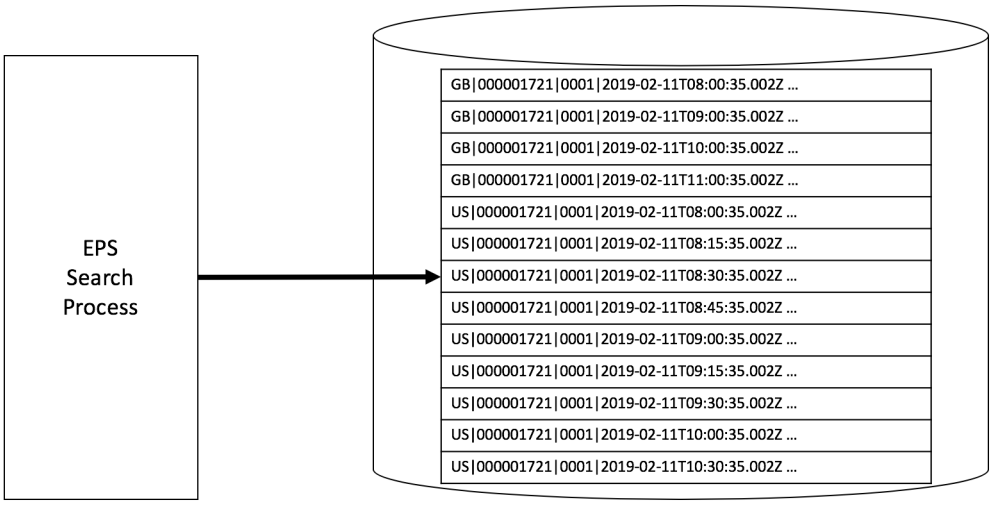



*Figure 4-2 Directly access relevant entries*

The EPS search process uses the **`EXEC CICS STARTBR`** command to begin browsing the records at this starting point. It examines this record and looks for other matching search criteria in the payload portion of the record. The process might find, for example, the producerID of `2e7d0f3f00d5acdc497b358779fa1b6f`). For any matches, it extracts the requested field values from the record and stores the result in memory.

It then issues an **`EXEC CICS READNEXT`** command to read the next record in the file and process it accordingly. This action is repeated sequentially, record by record. The command stops when it finds a record that contains a data/timestamp that is greater than the End date/time Range value that was provided in the search request. At this point, the service gathers the matching results from memory and returns them to the client.

### 4.1.3 Results

This approach yielded read operations of approximately 5,000 records per second. This is obviously, and unsurprisingly, well below the minimum required throughput. The inability to achieve the desired amount of reads with strictly sequential processing was anticipated, but this testing was necessary. It provided an initial baseline to be used in subsequent projections and testing.

Establishing this benchmark was useful, but it did prompt questions of how this single-threaded throughput might be increased. Although there was no expectation that any serial process would satisfy the needs of the application, this question leads to additional testing with less common mechanics.

## 4.2 Native VSAM

Additional testing was performed by using native VSAM/RLS operations, as opposed to the CICS browse APIs. This was done with the understanding that while low-level operations can improve efficiency by eliminating overhead, you forfeit valuable features and capabilities of the APIs. This section summarizes the results of this testing and some of the considerations that are given to this approach.

### 4.2.1 Results

The native VSAM/RLS operations were performed while running as THREADSAFE on an open task control block (TCB). With this method, we were able to process approximately 60,000 records per second. As expected, this is still not enough to satisfy the established minimum requirement, although it is substantially higher than the previous test. This benchmark proved to be useful for Walmart's use-case, but might not be meaningful in less extreme scenarios.

In the CICS open transaction environment (OTE), these entities can be defined to CICS as threadsafe: application programs, task-related user exits, global user exit programs, and user-replaceable modules. Then, they can run concurrently on open TCBs in the OTE, without causing wait issues on the quasi-reentrant task control block (QR TCB).

### 4.2.2 Tradeoffs

As mentioned in the opening statements of this section, taking this approach comes with costs. For example, ease of use is compromised to some degree. Of particular note is the loss of debugging capabilities in CICS APIs. In Walmart's case, feature flags for I/O types were incorporated into the programs. These flags allowed dynamic switching between native VSAM/RLS and EXEC CICS file control commands. These commands provide the many diagnostic capabilities of CICS APIs, such as **CEDF** and tracing. While this is helpful for some development activities, there are still compromises when you bypass the standard APIs.

### 4.2.3 Disclaimers

The intricacies of doing native VSAM/RLS I/O processing are beyond the scope of this publication. In fact, it is likely unnecessary in most processing scenarios. The Walmart use case that is discussed here is so demanding that it warrants exceptional measures. Consequently, the metric of 60,000 records per second serves as a reference point throughout the assessment in this document. Yet, this specific metric is not important to the overall context that this publication describes. The relevance of concepts related to asynchronous processing extend beyond the particular details in Walmart's situation.

## 4.3 Summary

Our initial tests established that sequential processing was not sufficient to achieve the necessary I/O rates for this use-case. However, the tests did provide valuable benchmarks for use in further iterations of a solution. The sequential read rates that we achieved were especially useful:

| | **CICS APIs** | **Native VSAM/RLS** |
|---|---|---|
| **Serial I/O rate per second** | 5,000 | 60,000 |

These sequential processing metrics might be adequate in some other less-demanding situations. Accommodations for these situations are briefly covered in Chapter 8. However, the need to satisfy the particular requirements of this application still existed, and as the saying goes, necessity is the mother of invention. Chapter 5, “Homegrown asynchronous solution” on page 23 explores the inventive homegrown solution that was developed to meet these objectives.

5

# Homegrown asynchronous solution

As described in Chapter 4, “Our initial sequential approach” on page 19, it quickly became apparent that serially searching the collection of events would not be sufficient. Our response time goals were still out of reach for our initial application. This chapter explores how we satisfied the SLA by using asynchronous methods to allow searches to run in parallel.

## 5.1 Parallel processing

The evolution of computer systems to multi-processor and multi-core systems led to the advent of parallelism in computing. Operations were split into smaller pieces that ran simultaneously on multiple processors or cores. This same concept can be applied to get around the constraints of sequential I/O rates in our scenario.

In Computer Science, Parallelism and Concurrency are similar concepts, but they are not identical. The general distinction is as follows:

- Parallelism involves literal simultaneous processing (in other words, each process on an individual core).
- Concurrency involves perceived simultaneous processing (in other words, time slicing of some of the processing).

In reality, both concepts can come into play for the solutions that are described in this document, depending on conditions. However, for the sake of simplicity, this document focuses on the Parallelism concept.

In Chapter 4, “Our initial sequential approach” on page 19, our preliminary tests established baseline rates for serial I/O in standard CICS APIs, then for native VSAM/RLS I/O operations.

We observed the rates that are listed in Table 5-1:

*Table 5-1 Sequential I/O Rates*

| | **CICS APIs** | **Native VSAM/RLS** |
|---|---|---|
| Serial I/O rate per second | 5,000 | 60,000 |

With Parallelism, you can achieve higher I/O rates than is afforded with single serial processes, which permits you to meet the prescribed processing goals. When multiple read tasks run simultaneously, the effective read rate becomes,

```
(the base serial read rate) X (the number of tasks)
```

Conversely, the number of parallel tasks that are needed to accommodate a particular throughput value can be calculated as,

```
(the total number of events) ÷ (the base serial read rate)
```

For example, Table 5-2 represents the number of parallel tasks that would be required to process 1 million events per second:

*Table 5-2 Required Parallel Tasks*

| **I/O Type** | **Total events** | **Serial read rate** (events per second) | **Parallel tasks** (Total events/Serial read rate) |
|---|---|---|---|
| CICS API | 1,000,000 | 5,000 | 200 |
| Native VSAM/RLS | 1,000,000 | 60,000 | 17 |

Looking at this another way, the total number of events processed per second by using parallel processing can be projected for a particular number of tasks for each I/O type. In the example in Table 5-3, 200 parallel tasks are assumed:

*Table 5-3 Max read rate at 200 tasks*

| **I/O Type** | **Serial read rate** (events per second) | **Task quantity** | **Effective read rate** (Serial read rate * Task quantity) |
|---|---|---|---|
| CICS API | 5,000 | 200 | 1,000,000 |
| Native VSAM/RLS | 60,000 | 200 | 12,000,000 |

These calculations suggest that the base requirement of processing 1 million events per second can be satisfied, and also that much higher throughputs can be achieved. However, managing parallelism like this presents many challenges. The number of tasks, the amount of work each task should do, the collation of results, and numerous other details must be considered. The following sections describe the mechanisms that we used to address these challenges.

## 5.2 Asynchronous processing

Parallel and asynchronous processing are admittedly two distinct concepts. But they can be used together. Specifically for this use-case, asynchronous processing mechanisms implement the parallelized I/O tasks. These tasks must be managed by some authoritative (or parent) process, and this process must be free (non-blocked) to manage multiple tasks within a specific request.

Additionally, search results from individual I/O tasks can be returned to the client in the order that they are processed by the service (in proper collation order, of course). That way, we avoid additional blocking to the client.

As covered in Chapter 2, “Background” on page 5, many techniques can be used to accomplish asynchrony in z/OS and CICS applications. The following section describes the techniques that are used by Walmart.

## 5.3 Design

As described in Chapter 3, “Requirements and challenges” on page 13, the search service must locate relevant events in the data store. Then it returns those events to the client in chronological order. Chapter 4, “Our initial sequential approach” on page 19 gave a description of the fundamental function of the search service, which still applies in this chapter. But now, we need the ability to split up and parallelize the I/O to the data store, so that we can process high volumes of entries within the response-time SLA. The remainder of this chapter focuses on the mechanics that accomplish this.

The following high-level steps show the sequence of events for parallel processing each search request:

1. Receive request from client
2. Use search parameters to identify and start I/O tasks
3. Monitor I/O tasks for completion
4. Gather and return results to client in the proper order

To accomplish these steps, the search service includes this task hierarchy:

- A parent task receives the client request and determines how many child tasks to start.
- The child tasks scour portions of the data store for relevant records and return those records to the parent task.

The parent task then returns the results to the client. This general design is depicted in Figure 5-1.

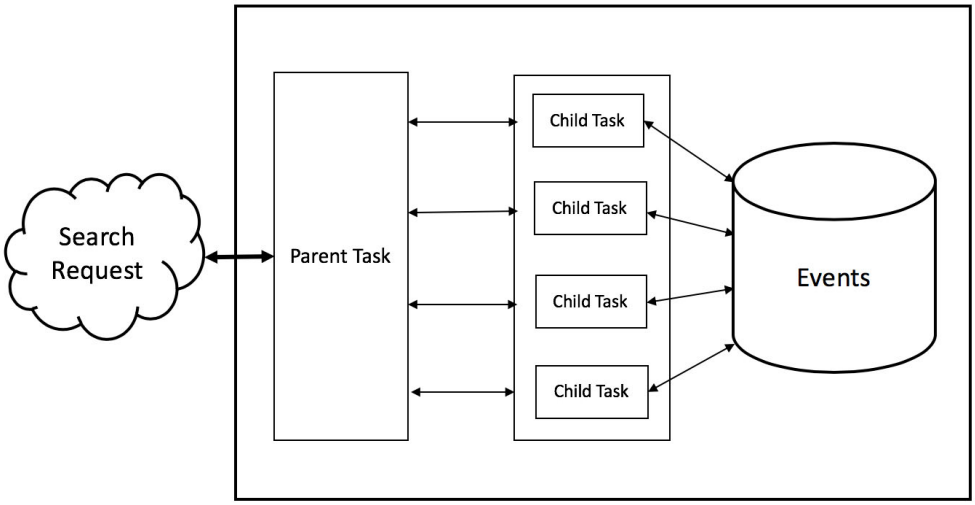



*Figure 5-1 High-Level Design*

Other components are needed to coordinate the activities between the main sections of this design. The focus for this document is the asynchronous mechanisms. So, these components are now explored from the perspective of a standard asynchronous design pattern, which includes the following activities:

- Prepare Data for Child
- Initiate Child
- Check for Completion
- Retrieve Data from Child
- Perform Housekeeping

### 5.3.1 Prepare Data for Child

An **HTTP GET** request from the client initiates the EPS search service. The **CICS WEB RECEIVE** and **WEB EXTRACT** commands are run to retrieve the search criteria. Then, control is passed to the parent program logic.

As discussed in Chapter 3, “Requirements and challenges” on page 13, each search request is predicated on a time range. Based on this time range, a proprietary algorithm determines the quantity and duration of intervals in the data to search. These intervals equate to the number of asynchronous child processes that the parent task initiates.

All events in the data store are time sequenced. Each child task must process a fraction of the overall time range of the search. As a result, each child task has values: a value for which a **STARTBR** command will be invoked, and a value for the final **READNEXT** value of its assigned range.

Figure 5-2 gives a simple view of the search request example from Chapter 4, “Our initial sequential approach” on page 19 that has been updated to reflect the multiple search intervals.

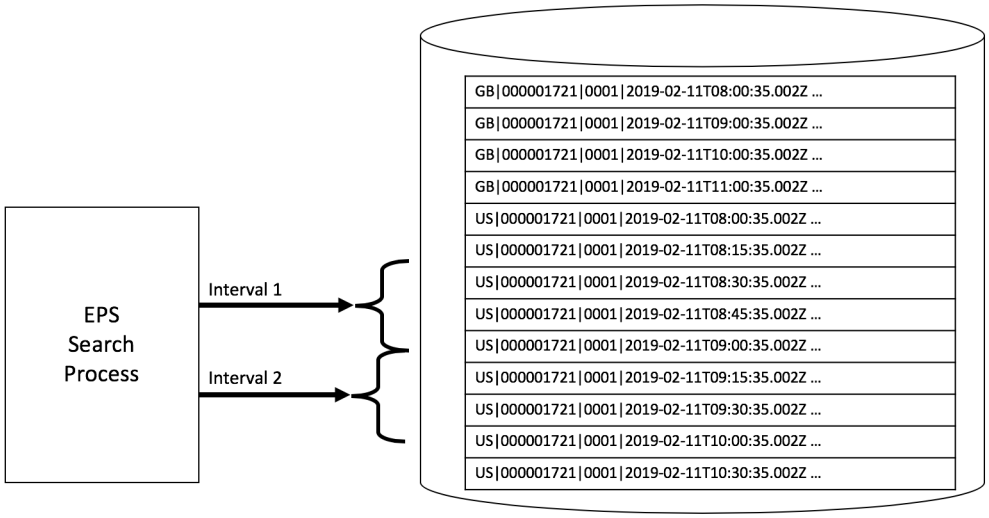



*Figure 5-2 Search intervals*

Because we want to retrieve the child tasks in chronological order, the parent task creates an array to store information about each child task. Various units of information are placed into this array and used by the parent task to manage the child tasks. Relevant components of the elements in this array include these items:

- Array index value

- Interval start and end times that are assigned to child
- Status flag/value

Sample code that shows the creation of the array is shown in Example 5-1.

*Example 5-1 Array in parent task*

```
************************************************************************ 00581362
* Start TSQ Table List.                                                * 00581489
* Maximum 255 entries.                                                 * 00581489
************************************************************************ 00581562
TT_DSECT DSECT
TT_START DS   0CL09
TT_S_HH  DS    CL02              TS Start HH
TT_S_MM  DS    CL02              TS Start MM
TT_S_SS  DS    CL02              TS Start SS
TT_S_MS  DS    CL03              TS Start MS (not part of TSQ name)
         DS    CL07              Align
*
TT_END   DS   0CL09              TS End   time                        00476045
TT_E_HH  DS    CL02              TS End   HH
TT_E_MM  DS    CL02              TS End   MM
TT_E_SS  DS    CL02              TS End   SS
TT_E_MS  DS    CL03              TS End   MS
         DS    CL07              Align
*
TT_209   DS   0CL09              TS Resume time                       00476045
TT_R_HH  DS    CL02              TS Resume HH
TT_R_MM  DS    CL02              TS Resume MM
TT_R_SS  DS    CL02              TS Resume SS
TT_R_MS  DS    CL03              TS Resume MS
         DS    CL03              Align
*
*
TT_STAT  DS    CL01              TS Table entry status
*                                A - active
*                                I - inactive
*                                R - resume (after 209)
*                                S - started
*                                C - completed
*
TT_IDX   DS    CL03              TS Index number
*
TT_E     EQU   *-TT_DSECT        Entry length
```

The parent task then uses its **EIBTRNID** and **EIBTASKN** values, along with the array index value of the associated child task to perform these actions (Example 5-2):

1. Create a unique channel name.
2. Store the relevant search request information for this child task through a **PUT CONTAINER**.
3. Repeat these steps for each child task that needs to be initiated.

*Example 5-2 TS and CHANNEL name in parent task*

```
************************************************************************ 00790110
* TSQ name for Child response.  The TSQ name will also serve as the   * 00790219
* CHANNEL name.  Child task uses the CHANNEL name as the TS response. * 00790219
```

```
************************************************************************ 00791019
         DS   0F
TS_TSQ   DS   0CL16              TS Queue name                           00476045
TS_TRAN  DS    CL04              TS Tran ID
TS_TASKN DS    CL07              TS Task Number
TS_IDX   DS    CL03              TS Index number
TS_SP    DS    CL02              TS Spaces
         DS   0F                                                        00475945
```

The updated design diagram in Figure 5-3 shows the addition of these components.

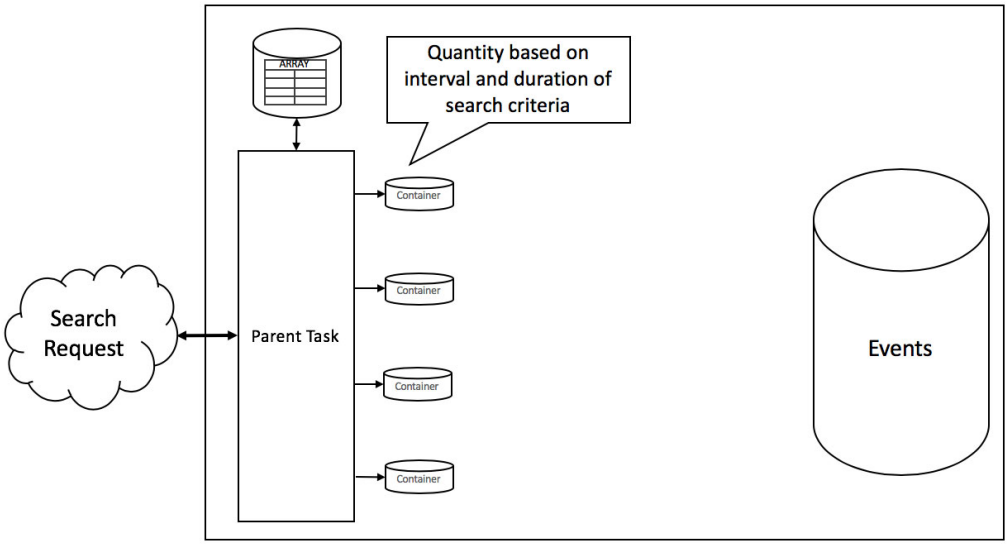



*Figure 5-3 Prepare Data for Child*

## 5.3.2 Initiate Child

After creating the set of request containers, the parent program issues a **START TRANSID** for each child task (Example 5-3). The **START TRANSID** commands include the **CHANNEL** option, which indicates the constructed name that corresponds with the array index for each particular child task. After the **START** commands run, the parent task updates the Status Flag portion of the array entry for each child to indicate that it has been started.

*Example 5-3 START TRANSID CHANNEL for parent task*

```
************************************************************************ 00790110
* Issue START for Child task providing the Channel name, which will   * 00790219
* be used as the TSQ response queue name.                             * 00790219
************************************************************************ 00791019
SY_0138  DS   0H                                                        00791110
         MVC   TS_CHILD,EIBTRNID       Move current TranID
         MVC   TS_CHILD+1,EC_CHILD     Move child Identifier
*
         MVI   TT_STAT,C'S'            Move 'started' indicator
         MVC   TS_IDX,TT_IDX           Move index number
*
         EXEC CICS START                                               X
              TRANSID   (TS_CHILD)                                     X
              CHANNEL   (TS_TSQ)                                       X
```

```
              NOHANDLE
*                                                                       00791223
```

When a child task starts, an **ASSIGN CHANNEL** command runs and gets the channel name that was created by the parent task. The child task then uses this name to create a Temporary Storage (TS) queue into which it posts response information (Example 5-4).

*Example 5-4 ASSIGN CHANNEL in child task*

```
*********************************************************************** 00790110
* Issue ASSIGN for CHANNEL name, which is used as the TS response.    * 00790219
*********************************************************************** 00791019
SY_0000  DS   0H                                                        00791110
         EXEC CICS ASSIGN CHANNEL(CHANNEL) NOHANDLE
         MVC   TS_QNAME,CHANNEL        Move CHANNEL name to TSQNAME
*
```

The child task issues the **GET CONTAINER** command to acquire the search request information that was passed from the parent task. Then the child task performs a **GETMAIN SHARED** operation to establish a location to store the result set from its search assignment (Example 5-5).

*Example 5-5 Issue GETMAIN SHARED in child task*

```
*********************************************************************** 01070292
* Issue GETMAIN for Result Set in SHARED storage                      * 01070392
*********************************************************************** 01070292
GM_0020  DS   0H                                                         00973499
         ST    R14,GM_REG              Save return register             01070893
*                                                                        00806542
         EXEC CICS GETMAIN                                              X00806642
               SET(R1)                                                  X00806764
               FLENGTH(G_LENGTH)                                        X00806864
               INITIMG(HEX_00)                                          X00806986
               SHARED                                                   X00806986
               NOHANDLE                                                  00806986
*                                                                        00806542
         L     R14,GM_REG              Load return register             01070893
         BCR   B'1111',R14             Return to caller                 01070893
*                                                                        00948799
```

Then, the child task issues the **STARTBR** (Example 5-6) and **READNEXT** (Example 5-7) commands until it has processed its entire assigned interval.

*Example 5-6 STARTBR for parallel I/O child task*

```
*********************************************************************** 00790110
* Issue STARTBR on Primary Column Index when this service is not      * 00790219
* defined to the ECM zPARM as HP I/O 'yes'.                           * 00790219
*********************************************************************** 00791019
SY_0085  DS   0H
*
         EXEC CICS STARTBR                                              X
              FILE     (WF_FCT)                                         X
              RIDFLD   (DF_KEY)                                         X
              GTEQ                                                      X
```

```
             NOHANDLE
*
         CLC   EIBRESP,=F'13'          NOTFND  condition?
         BRC   B'1000',ER_20401        ... yes, STATUS(204)
         OC    EIBRESP,EIBRESP         Normal condition?
         BC    B'0111',ER_50701        ... no,  File I/O error
*
************************************************************************ 00790110
* GET for HP I/O or READNEXT for API method.                           * 00790219
************************************************************************ 00791019
SY_0090  DS   0H
         CLI   WS_HPIO,C'Y'            ECM HP I/O enabled?
         BRC   B'0111',SY_0093         ... no,  use EIP services
*
         CLI   HP_STAT,C'Y'            HP I/O active?
         BRC   B'0111',ER_50712        ... no,  exit stage left.
```

*Example 5-7 READNEXT for parallel I/O child task*

```
************************************************************************ 00790110
* Issue READNEXT until EOF or key range is exceeded.                   * 00790219
************************************************************************ 00791019
SY_0093  DS   0H
         MVC   WF_LEN,=H'32700'        Move record length
         L     R10,FF_ADDR             Load record address
*
         EXEC CICS READNEXT                                            X
              FILE  (WF_FCT)                                           X
              RIDFLD(DF_KEY)                                           X
              INTO  (DF_DATA)                                          X
              LENGTH(WF_LEN)                                           X
              NOHANDLE
*
         CLC   EIBRESP,=F'20'          ENDFILE condition?
         BRC   B'1000',SY_0899         ... yes, set EOF
         CLC   EIBRESP,=F'13'          NOTFND  condition?
         BRC   B'1000',SY_0899         ... yes, set EOF
*
         OC    EIBRESP,EIBRESP         Normal condition?
         BRC   B'0111',ER_50702        ... no,  File I/O error
*
```

An additional container is used during this process and it has the other search criteria for fields in the payload section of the records. This container is not relevant to the asynchronous mechanisms that we are discussing. It is omitted from this design description to maintain simplicity and avoid confusion.

After the child task processes and stores all results for its assigned interval, the task runs a **WRITEQ TS** operation that uses the name that was obtained from the earlier **ASSIGN CHANNEL** command (Example 5-8).

*Example 5-8 Issue WRITEQ TS for response in child task*

```
************************************************************************ 01111599
* Put response information in TSQ for Parent task to process.          * 01112099
```

```
*********************************************************************** 01120010
TS_0030  DS   0H                                                        00791110
         EXEC CICS WRITEQ TS                                           X
              QNAME   (TS_QNAME)                                       X
              FROM    (TS_REC)                                         X
              ITEM    (TS_ITEM)                                        X
              LENGTH  (TS_LEN)                                         X
              MAIN                                                     X
              NOHANDLE
*                                                                       01077147
TS_0099  DS   0H                                                        00791110
         L     R14,TS_REG             Load return register             01070893
         BCR   B'1111',R14            Return to caller                 01070893
*                                                                       01077147
```

The data that is written to the TS queue includes status information and the address of the result set that is in the **GETMAIN SHARED** area. The updated design diagram in Figure 5-4 shows the addition of these components.

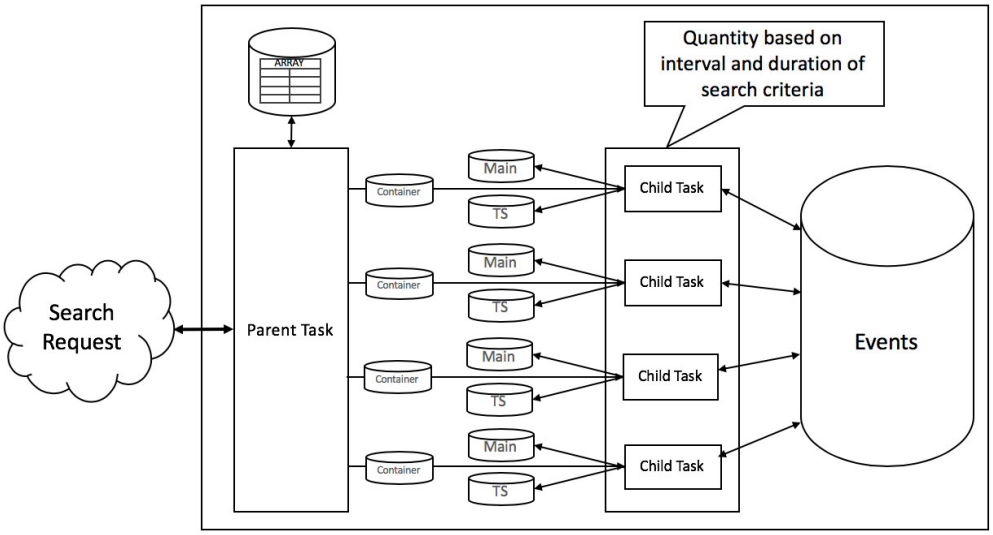



*Figure 5-4 Initiate Child*

## 5.3.3 Check for Completion

After all required child tasks are instantiated, the parent task begins to process the responses. The results must be returned to the client in chronological order, so the internal array is used to sequence the processing of responses.

As mentioned in the previous section, a child task completes its processing by issuing a **WRITEQ TS** command. This command creates a uniquely-named TS queue that is derived from information in the management array, which was created by the parent task. The information in the TS queue name includes the array index value that is associated with the corresponding child task. The parent task uses this information to issue a **READQ TS** command to that unique TS queue name.

When the TS queue does not exist, this condition indicates that the child task has not completed. In this case, the parent task issues a **STIMERM** macro (SVC 47) with a default of 50 milliseconds. Then the parent task branches back to the **READQ TS** and attempts to process

that child response again. It repeats this process until it receives a response from that child task. Then the parent task proceeds through the remaining entries in the array using the same method. If a total processing time of 30 seconds is reached, the request is ended. See Example 5-9 on page 32.

*Example 5-9 Synchronicity in parent task*

```
************************************************************************ 00790110
* Started entry found.  Issue READQ for the TS_TSQ name.              * 00790219
* If the TSQ is not available, issue a STIMERM for 50ms and continue  * 00790219
* this cycle for 600 times (30 seconds), then issue a Time-Out.       * 00790219
************************************************************************ 00791019
SY_0220  DS   0H
         LA    R1,TS_L                 Load TSQ record length
         STH   R1,TS_LEN               Save TSQ record length
         MVC   TS_IDX,TT_IDX           Move TSQ index number
*
         EXEC CICS READQ TS                                           X00791223
              QNAME (TS_TSQ)                                          X00791223
              INTO  (TS_REC)                                          X00791223
              LENGTH(TS_LEN)                                          X00791223
              ITEM  (TS_ITEM)                                         X00791223
              NOHANDLE                                                 00791223
*
         OC    EIBRESP,EIBRESP         Normal response?
         BRC   B'1000',SY_0230         ... yes, continue process
*
         L     R1,SM_COUNT             Load STIMERM count
         LA    R1,1(,R1)               Add 1
         ST    R1,SM_COUNT             Save STIMERM count
         C     R1,SM_MAX               Max  STIMERM time?
         BRC   B'1011',SY_0282         ... yes, log a Time-Out
*
************************************************************************ 00791019
* STIMERM Macro does not support relative addressing, so I'm coding   * 00790219
* the instructions with the necessary adjustments.                    * 00790219
************************************************************************ 00791019
         OC    MS_WAIT,MS_WAIT         Wait set already?
         BRC   B'0111',*+10            ... yes, bypass default
         MVC   MS_WAIT,=F'5'           ... no,  set 50 ms to interval
         LA    R8,STIMERID             Load STIMER ID
         LA    R9,MS_WAIT              Load wait time
*
*        STIMERM SET,BINTVL=(R9),WAIT=YES,ID=(R8)
*
SY_0225  DS   0H                       STIMERM invocation
         LAE   R1,SM_LIST              Set up list address
         MVC   0(4,R1),=X'11000001'    Flag byte and LVL#
         ST    R8,4(,R1)               Store ID address in SM_LIST
         ST    R9,8(,R1)               Store Interval address in list
         LA    0,4                     Load Option byte into R0
         SLL   0,24                    Shift Option (bit 5 on)
         SVC   47                      Issue STIMERM SET SVC
*
         BRC   B'1111',SY_0220         Continue READQ for same Child
```

*

The updated design diagram in Figure 5-5 shows the objects that are related to the process of checking for completion.

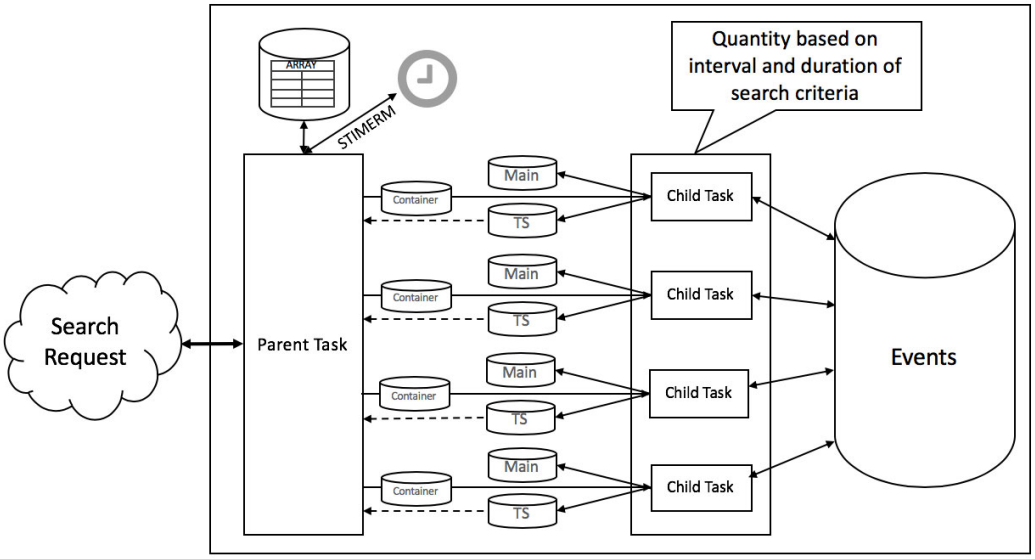



*Figure 5-5 Check for Completion*

## 5.3.4 Retrieve Data from Child

After the **READQ TS** queue runs successfully, the address of the result set (a **GETMAIN SHARED** address) is obtained from the TS queue data. This result set from the child is sent to the client by using chunked message transfer through a **WEB SEND** command. After the **WEB SEND** command is complete, the parent increments the index by one. Then the parent processes the next array entry and repeats this process until the last child response has been sent to the client. The updated design diagram in Figure 5-6 shows these additional data-pull relationships.

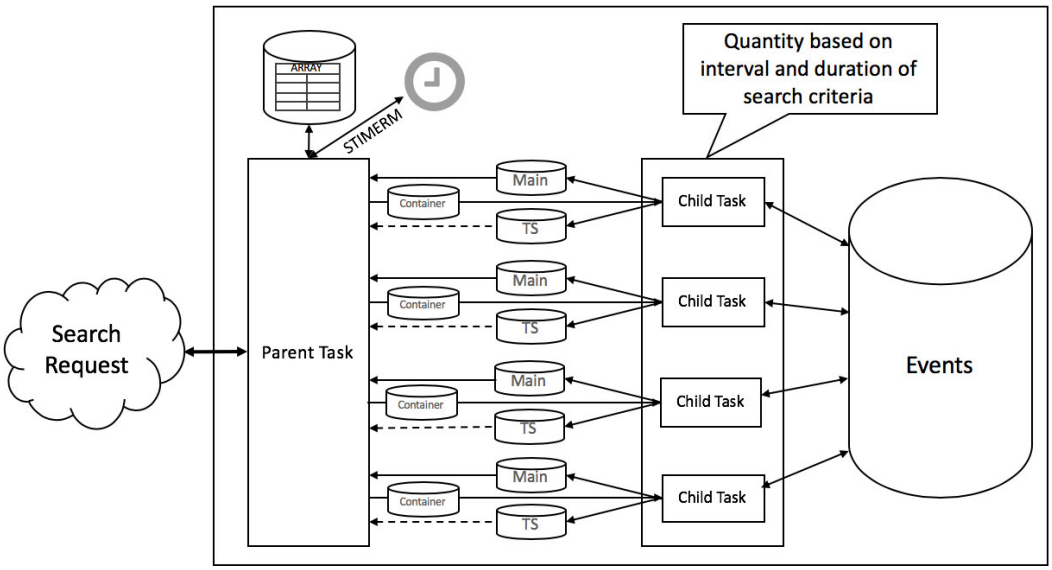



*Figure 5-6 Retrieve Data from Child*

## 5.3.5 Perform Housekeeping

Along with managing client requests, task coordination, and response processing, the service must also do resource management. In particular, the service supervises and properly reclaims the various types of storage that are employed in this design. This process can be quite complex. When handled improperly, this process might cause storage to be orphaned, which has a negative impact on both the service and the CICS region or system.

Even under normal circumstances, TS queues and **GETMAIN SHARED** storage areas are not released or freed when the parent task or the child task terminates. In this case, extra logic for housekeeping is necessary in the parent task. After it completes the processing of each child task response, the parent task must run a **DELETEQ TS** operation to clean up the TS queue (Example 5-10). Then it runs **FREEMAIN** to release storage that was directly obtained by the child process (Example 5-11).

*Example 5-10 DELETEQ of response TS queue in parent task*

```
************************************************************************ 01070292
* Issue DELETEQ TS for Child TS queue                                  * 01070392
************************************************************************ 01070292
TS_0010  DS   0H                                                         00973499
         ST    R14,TS_REG             Save return register               01070893
*                                                                        00806542
         EXEC CICS DELETEQ TS                                           X00806642
               QNAME(TS_TSQ)                                            X00806764
               NOHANDLE                                                  00806986
*                                                                        00806542
         L     R14,TS_REG             Load return register               01070893
         BCR   B'1111',R14            Return to caller                   01070893
```

*Example 5-11 FREEMAIN of SHARED storage in parent task*

```
************************************************************************ 00790110
* Issue FREEMAIN for Response Array buffer                             * 00790219
```

```
* Check for resume status.                                            * 00790219
********************************************************************** 00791019
SY_0280  DS   0H                                                        00791110
         MVC   TS_IDX,TT_IDX           Move index number
         L     R3,TS_RA_A              Load Child message buffer
         BRAS  R14,FM_0010             Issue FREEMAIN
*
         CLI   TT_STAT,C'R'            Resume interval?
         BRC   B'0111',SY_0290         ... no,  get next entry
*
         LH    R1,TS_ITEM              Load Item Number
         A     R1,=F'1'                Add 1 to Item Number
         STH   R1,TS_ITEM              Save Item Number
         BRC   B'1111',SY_0220         ... yes, process same entry
*
********************************************************************** 01070292
* Issue FREEMAIN for Child message buffer                             * 01070392
********************************************************************** 01070292
FM_0010  DS   0H                                                        00973499
         ST    R14,FM_REG              Save return register            01070893
         LTR   R3,R3                   Zero address?
         BRC   B'1000',FM_0099         ... yes, bypass FREEMAIN
*                                                                       00806542
FM_0020  DS   0H                                                        00973499
         EXEC CICS FREEMAIN                                            X00806642
               DATAPOINTER(R3)                                         X00806764
               NOHANDLE                                                 00806986
*                                                                       00806542
FM_0099  DS   0H                                                        00973499
         L     R14,FM_REG              Load return register            01070893
         BCR   B'1111',R14             Return to caller                01070893
*                                                                       00948799
```

However, abnormal conditions must also be considered. Any premature termination of the service might also orphan storage and lead to instability. To address this risk, another level of housekeeping is incorporated into the design.

An independent background task is defined as an Interval Control Element (ICE) to run periodically for each service. As described earlier in this chapter, the **EIBTRNID** and **EIBTASKN** values are used to establish unique channel names that tasks can use. These names are used in TS queue definitions. This information also is used by the background housekeeping process. The background task takes the following actions:

- Issue **INQUIRE TSQUEUE START** and **NEXT** commands to browse TS queues.
- Check **EIBTRNID** value to identify associated service instance.
- Use the **EIBTASKN** value on an **INQUIRE TASK** command to determine whether parent task is active.
- If parent task is no longer active,
  - Issue **READQ TS** against queue name to get **GETMAIN SHARED** address.
  - Issue **FREEMAIN** to release storage.
  - Issue **DELETEQ TS** to release TS queue.

This process adds even more components to the design. The updated design diagram in Figure 5-7 shows these additional parts.

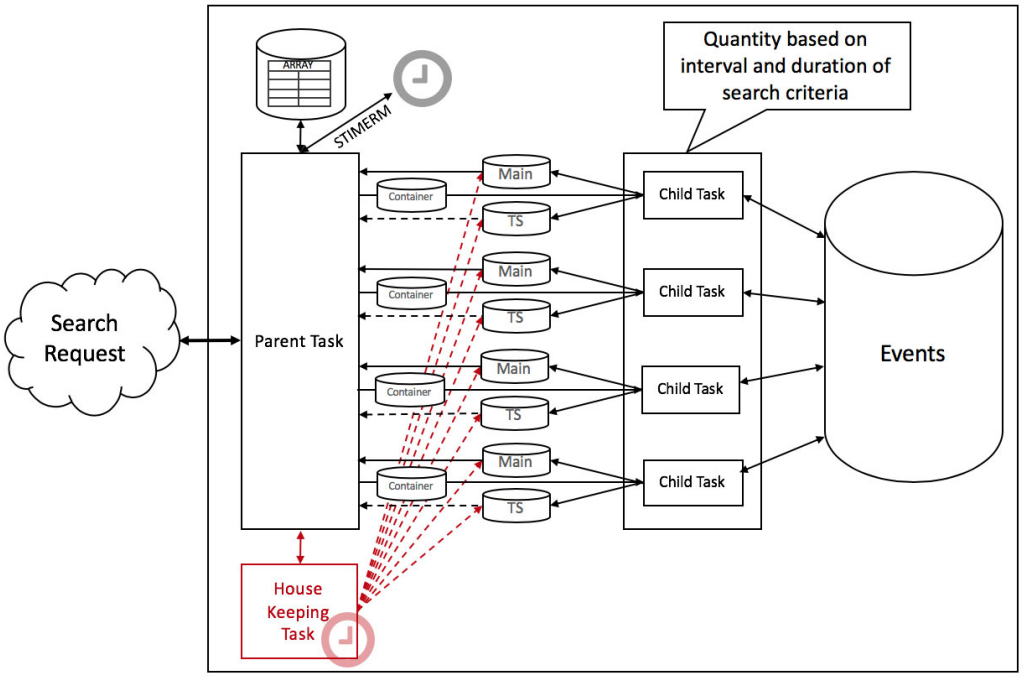



*Figure 5-7 Retrieve Data from Child*

## 5.4 Summary

This chapter has described the main components and provided high-level views of the original design. This design achieves parallelism with asynchronous methods to achieve the I/O rates that the application requires. The projections of throughput rates that can be achieved by parallelizing the search activity held true. The objective of processing at least 1 million events per second was accomplished.

Even with the simplified description of the design, this chapter reveals the complexity of the solution. In Chapter 6, “IBM CICS asynchronous solution” on page 37, the same general design is described, but it is based on CICS asynchronous API instead of custom-built mechanics.

6

# IBM CICS asynchronous solution

This chapter explores the version of the search function that uses the IBM CICS asynchronous API to perform parallel I/O processing. We also describe the actions that were needed to migrate from the homegrown solution to this new design.

> "The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it." --Steve Jobs, Apple

The preceding quotation reflects the complexity and difficulty of programming for parallelism. The complexity of the homegrown solution in Chapter 5, “Homegrown asynchronous solution” on page 23 illustrates this fact. The design in this chapter demonstrates how the CICS asynchronous API reduces that difficulty.

## 6.1 Design

The basic sequence of events for processing each search request is unchanged from what was described in 5.3, “Design” on page 25 and is repeated here:

1. Receive request from client
2. Use search parameters to identify and start I/O tasks
3. Monitor I/O tasks for completion
4. Gather and return results to client in the proper order

Also, the same general components are used to accomplish these steps, including this:

- A parent task that receives the client request and determines how many child tasks to start.
- The child tasks that search sections and return relevant results to the parent task.

The parent task will then return the results to the client. This general design is depicted in Figure 6-1

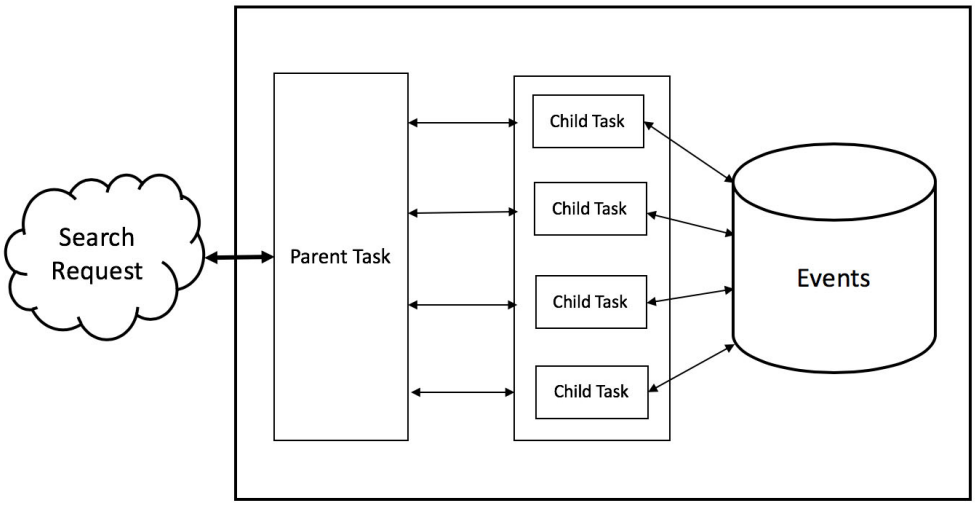



*Figure 6-1 High-Level Design*

The simplification that the CICS asynchronous API brings becomes apparent when we explore the details of the design components. These details are highlighted in the following sections. Like Chapter 5., “Homegrown asynchronous solution” on page 23, these details include the following general activities of a standard asynchronous pattern:

- Prepare Data for Child
- Initiate Child
- Check for Completion
- Retrieve Data from Child
- Perform Housekeeping

### 6.1.1 Prepare Data for Child

Only slight changes are needed for this stage of the process, as compared to the homegrown solution. The ingest handling and fundamental steps for child data preparation that are described in 5.3.1, “Prepare Data for Child” on page 26 are the same and can be summarized as follows:

1. The **`WEB RECEIVE`** and **`WEB EXTRACT`** commands retrieve search criteria from the client **`HTTP GET`** request, and control is passed to the parent program.
2. Based on the total time range of the search request, a proprietary algorithm determines the quantity and duration of intervals to search, and therefore the number of child tasks to prepare.
3. The parent task creates an in-memory array that includes various bits of identifying information that are used to manage each child task.
4. The parent task uses this identifying information (along with its **`EIBTRNID`** and **`EIBTASKN`** values) to create a unique channel name. Then it stores search criteria by using a **`PUT CONTAINER`** command for each child task.

From this perspective, the high-level design is similar to the homegrown solution as it was previously depicted in Chapter 5., “Homegrown asynchronous solution” on page 23.

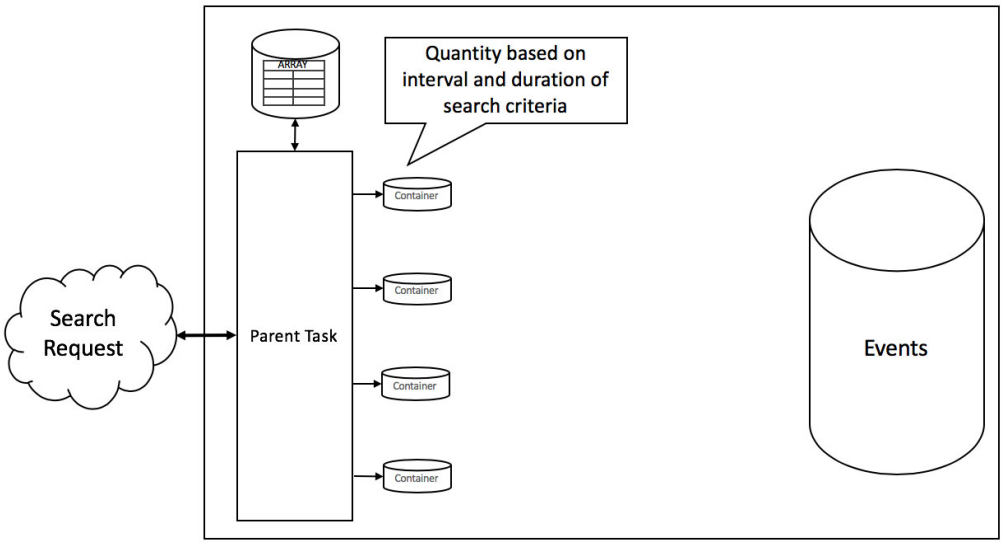



*Figure 6-2 Prepare Data for Child*

However, when you use the CICS asynchronous API, there is one key difference to this section of the solution. That difference is related to the array that contains identifying information for each child task. Now, the array also includes the child token and container name for each entry. The array must be defined to accommodate these values, which become available at child initiation. This information is used by various invocations of the asynchronous API in subsequent stages of the process. See Example 6-1.

*Example 6-1 Array in parent task*

```
********************************************************************** 00581362
* Start TSQ Table List.                                              * 00581489
* Maximum 255 entries.                                               * 00581489
********************************************************************** 00581562
TT_DSECT DSECT
TT_CHILD DS    CL16              Child Token    (RUN TRANSID)
TT_CHAN  DS    CL16              Child channel  (FETCH)
TT_STAT  DS    F                 Child COMPSTAT (FETCH)
TT_RESP  DS    F                 Child Response (FETCH)
TT_T_OUT DS    F                 Child Timeout  (FETCH)
*
TT_START DS   0CL09
TT_S_HH  DS    CL02              TS Start HH
TT_S_MM  DS    CL02              TS Start MM
TT_S_SS  DS    CL02              TS Start SS
TT_S_MS  DS    CL03              TS Start MS (not part of TSQ name)
         DS    CL07              Align
*
TT_END   DS   0CL09              TS End   time                        00476045
TT_E_HH  DS    CL02              TS End   HH
TT_E_MM  DS    CL02              TS End   MM
TT_E_SS  DS    CL02              TS End   SS
TT_E_MS  DS    CL03              TS End   MS
         DS    CL07              Align
*
TT_209   DS   0CL09              TS Resume time                       00476045
```

```
TT_R_HH  DS    CL02              TS Resume HH
TT_R_MM  DS    CL02              TS Resume MM
TT_R_SS  DS    CL02              TS Resume SS
TT_R_MS  DS    CL03              TS Resume MS
         DS    CL03              Align
*
*
TT_STAT  DS    CL01              TS Table entry status
*                                A - active
*                                I - inactive
*                                R - resume (after 209)
*                                S - started
*                                C - completed
*
TT_IDX   DS    CL03              TS Index number
*
TT_E     EQU   *-TT_DSECT        Entry length
```

### 6.1.2 Initiate Child

This stage of the process includes more substantial changes. This stage of the homegrown solution included these initial actions:

1. Issuing a **START TRANSID** with the **CHANNEL** option for each child task.
2. Upon completion of each **START TRANSID** command, the parent task updates a status flag in the control array.

Now, in the asynchronous API solution this process is replaced with the following actions:

1. Issue a **RUN TRANSID** with the **CHANNEL** option for each child task (Example 6-2).
2. Upon completion of each **RUN TRANSID** command, the parent task updates a status flag in the control array.
3. The array is also updated with the token from the **CHILD** parameter of the **RUN TRANSID** command.

*Example 6-2 RUN TRANSID CHANNEL for parent task*

```
********************************************************************** 00790110
* Issue RUN   for Child task providing the Channel name, which will   * 00790219
* be used to provide request CONTAINER information.                   * 00790219
********************************************************************** 00791019
SY_0138  DS   0H                                                       00791110
         MVC   PP_TRAN,EIBTRNID        Move current TranID
         MVC   PP_TRAN+1,EC_CHILD      Move child Identifier
*
         MVI   TT_COMP,C'S'            Move 'started' indicator
         MVC   PP_IDX,TT_IDX           Move index number
*
         EXEC CICS RUN                                                X
              TRANSID   (PP_TRAN)                                     X
              CHILD     (PP_CHILD)                                    X
              CHANNEL   (PP_CHAN)                                     X
              NOHANDLE
*                                                                      00791223
         MVC   TT_CHILD,PP_CHILD       Move Child token to table
```

As before, the channel option specifies a name that corresponds with the array index of the particular child task. And the status flag is used by the parent task to manage the child tasks. The value of the child token is used to simplify some of the remaining steps of the process.

In the homegrown solution, each child task ran an **ASSIGN CHANNEL** command to obtain the unique channel name that was created by the parent task for that particular child. That name was used by the child task to create a TS queue name for response status information. This technique is eliminated in the asynchronous API solution. Instead, a **PUT CONTAINER** command is used to achieve this purpose. The **GETMAIN SHARED** is also replaced with a local (non-shared) **GETMAIN** and another **PUT CONTAINER** operation.

In the new solution, the child task issues a **GET CONTAINER** to acquire the search criteria from the parent task. Then it issues a local **GETMAIN** to house its result set. Then, each child task runs **STARTBR** and **READNEXT** commands, which remain identical to the original process. These actions peruse the records in the child task's assigned interval. Any matches that it finds are placed into the **GETMAIN** storage

After the child task processes its assigned interval, the two **PUT CONTAINER** commands run (Example 6-3):

- One **PUT CONTAINER** provides status information to the parent task.
- The other **PUT CONTAINER** passes the result set from the local **GETMAIN** to the parent task.

*Example 6-3 Issue PUT CONTAINER for response in child task*

```
************************************************************************ 01111599
* Put response information in Container for Parent task to process.   * 01112099
************************************************************************ 01120010
TS_0030  DS   0H                                                         00791110
         EXEC CICS PUT                                                  X
              CONTAINER(C_STATUS)                                       X
              CHANNEL  (PP_CHAN)                                        X
              FROM     (PP_REC)                                         X
              FLENGTH  (PP_FLEN)                                        X
              NOHANDLE
*                                                                        01077147
         CLC   PP_CODE,=C'204'         Status 204?
         BRC   B'1000',TS_0099         ... yes, no response set
         L     R4,RA_ADDR              Load response array address
         USING RA_DSECT,R4             ... tell assembler
*                                                                        01077147
         EXEC CICS PUT                                                  X
              CONTAINER(C_RESULT)                                       X
              CHANNEL  (PP_CHAN)                                        X
              FROM     (RA_DSECT)                                       X
              FLENGTH  (RA_LEN)                                         X
              NOHANDLE
*                                                                        01077147
         DROP  R4                      ... tell assembler
*                                                                        01077147
TS_0099  DS   0H                                                         00791110
         L     R14,TS_REG              Load return register             01070893
         BCR   B'1111',R14             Return to caller                 01070893
*
```

These changes make a design that looks very similar to the original homegrown solution. But the management of differing storage constructs is now simplified by the use of containers. This is reflected in Figure 6-3.

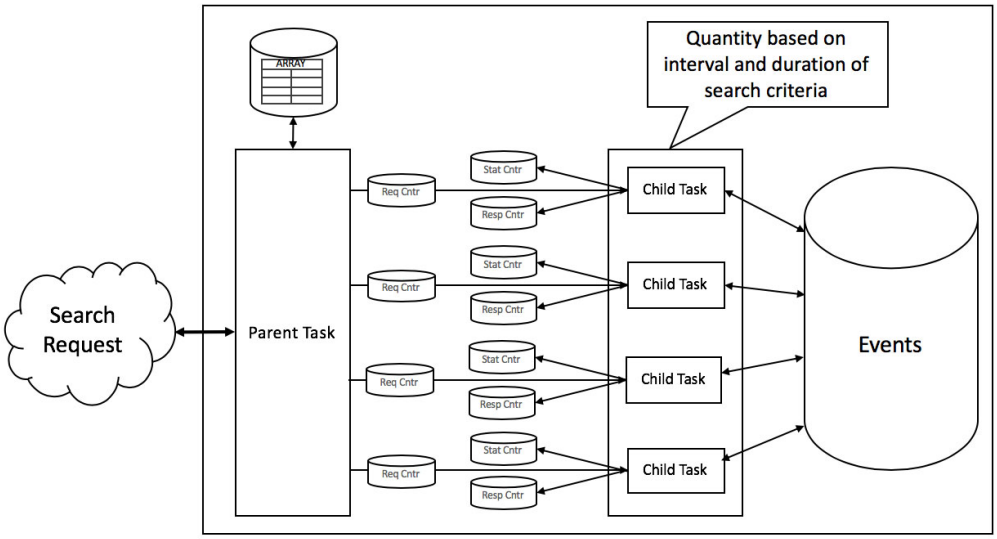



*Figure 6-3 Initiate Child*

## 6.1.3 Check for Completion

The basic approach for this step is similar to the homegrown solution. After all child tasks are started, the parent program begins attempting to process the response from each child task in the order they appear in the array.

However, the details of how this is accomplished are quite different. The techniques from the homegrown solution are almost entirely replaced with CICS asynchronous API commands.

Following the order of tasks in the array, a **FETCH CHILD** command is issued. The command uses the child token that was captured when each particular task was run. Upon successful response from CICS (and subsequent data retrieval), this action is simply repeated with the child token of the next item in the array. The **READQ TS** and the **STIMERM**-based polling loop have been eliminated.

An overall completion time of all tasks must still be considered. The **TIMEOUT** parameter of the **FETCH CHILD** command is used for this purpose. The parent task uses a simple internal timer to set the timeout values. The pre-defined timeout for the service is 30 seconds. So, the task uses this value for the **TIMEOUT** parameter on the initial **FETCH CHILD**. Then, it decrements the value (based on the internal timer) on each iteration of the process. This approach ensures that the cumulative processing is accounted for when a request times out after running for a long time. See Example 6-4.

*Example 6-4 Synchronicity in parent task*

```
************************************************************************ 00790110
* Issue FETCH for the TT_CHILD token and specify a TIMEOUT default of * 00790219
* 30 seconds.  Set status 534 for TIMEOUT, then end the request.      * 00790219
* Resources for the FETCH will be stored in the current TT_DSECT      * 00790219
* array index.                                                        * 00790219
* The CHANNEL returned from the FETCH will be used for subsequent     * 00790219
* GET CONTAINER commands to retrieve result set and response status   * 00790219
```

```
* from the CHILD task.                                                 * 00790219
********************************************************************** 00791019
SY_0220  DS   0H
         OC    TT_T_OUT,TT_T_OUT       Time-out already set?
         BRC   B'0111',*+10            ... yes, bypass default
         MVC   TT_T_OUT,=F'30000'      ... no,  set 30 second default
*
         EXEC CICS FETCH                                              X
              CHILD     (TT_CHILD)                                    X
              CHANNEL   (TT_CHAN)                                     X
              COMPSTATUS(TT_STAT)                                     X
              RESP      (TT_RESP)                                     X
              TIMEOUT   (TT_T_OUT)                                    X
              NOHANDLE
*
         L     R1,EC_C_SEC             Load current seconds
         S     R1,EC_S_SEC             Subtract starting seconds
         C     R1,SM_MAX               Max time exceeded?
         BRC   B'1010',SY_0282         ... yes, set HTTP status 534
*
         CLC   TT_RESP,DFHRESP(NOTFINISHED)     Timeout?
         BRC   B'1000',SY_0282         ... yes, set HTTP status 534
*
         CLC   TT_STAT,DFHVALUE(NORMAL)         Normal response?
         BRC   B'0111',ER_50303        ... no,  set HTTP status 503
*
         LA    R1,PP_L                 Load PP  record length
         ST    R1,PP_FLEN              Save PP  record length
         MVC   PP_IDX,TT_IDX           Move PP  index number
*
         EXEC CICS GET                                               X00791223
              CONTAINER(C_STAT)                                      X00791223
              CHANNEL  (TT_CHAN)                                     X00791223
              INTO     (PP_STAT)                                     X00791223
              FLENGTH  (PP_FLEN)                                     X00791223
              NOHANDLE                                                00791223
*
         OC    EIBRESP,EIBRESP         Normal response?
         BRC   B'0111',ER_50304        ... no,  set HTTP status 503
*
         EXEC CICS GET                                               X00791223
              CONTAINER(C_RESULT)                                    X00791223
              CHANNEL  (TT_CHAN)                                     X00791223
              SET      (R4)                                          X00791223
              FLENGTH  (PP_FLEN)                                     X00791223
              NOHANDLE                                                00791223
*
         OC    EIBRESP,EIBRESP         Normal response?
         BRC   B'0111',SY_0284         ... no,  must be 204
*
         ST    R4,PP_RA_A              Save response array address
         MVC   PP_RA_L,PP_FLEN         Save response array length
         BRC   B'1111',SY_0230         Continue process
```

No additional items are added to the high-level design with this solution because there is no need to read TS queues or add a polling mechanism. Figure 6-4 illustrates this fact.

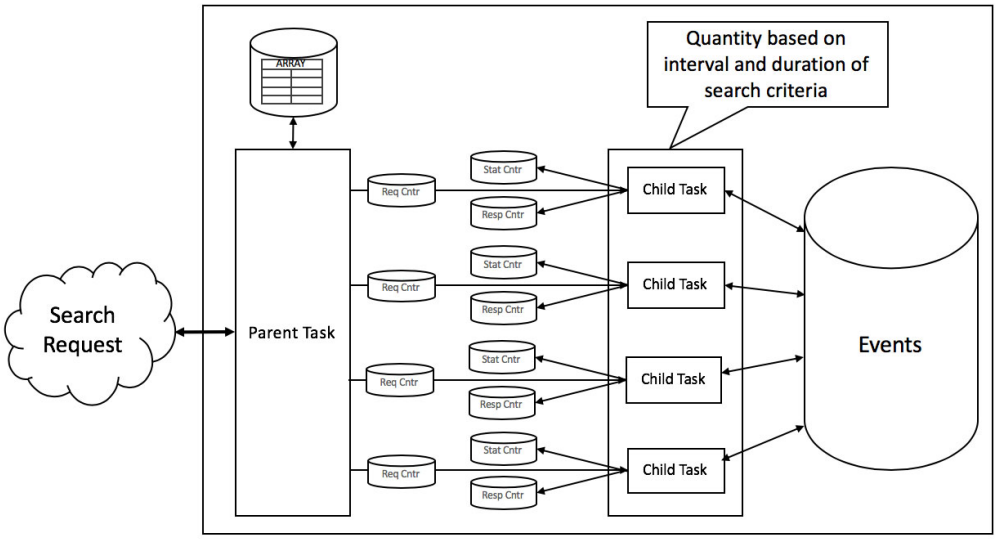



*Figure 6-4 Check for Completion*

## 6.1.4 Retrieve Data from Child

The parent no longer uses the **READQ TS** or **GETMAIN** operations to retrieve data from the child tasks in the new solution. Upon completion of the **FETCH CHILD** command, the parent task simply issues these commands:

- A **GET CONTAINER** for the status information.
- Another **GET CONTAINER** for the result set that the child task provides.

From this point, the actions are the same as the homegrown solution. The parent task uses chunked message transfer through the **WEB SEND** command to return the result set to the client. It then goes to the next entry in the array and processes the next child task. This activity is repeated until all child tasks have been processed. Relationships for the **GET CONTAINER**s are added to the high-level diagram in Figure 6-5.

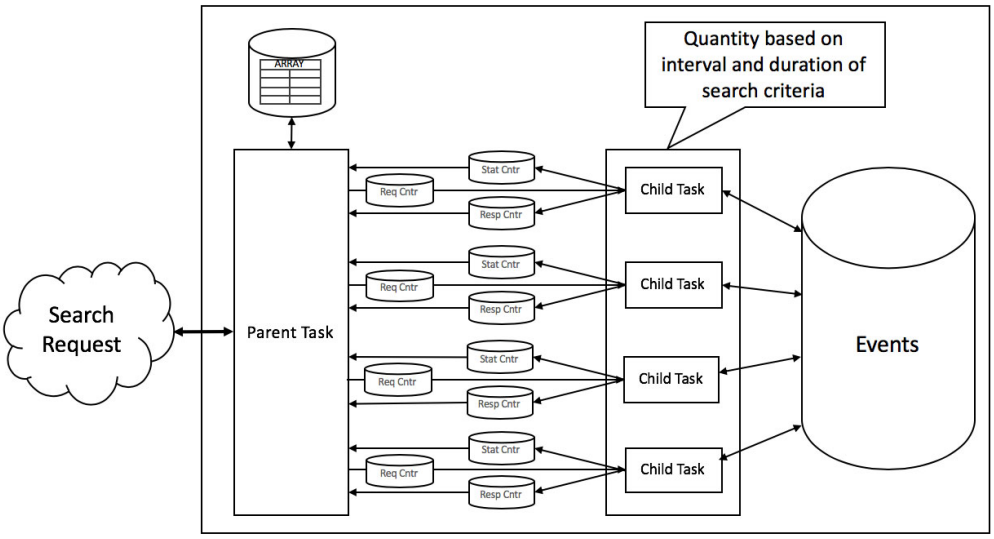



*Figure 6-5 Check for Completion*

## 6.1.5 Perform Housekeeping

The CICS asynchronous API brought substantial reductions in complexity to the housekeeping phase. A large part of the homegrown solution is simply removed. The parts that remain are simplified considerably.

In the homegrown solution, the parent task issues **DELETEQ TS** and **FREEMAIN** commands to clean up the storage that was used for status responses and result sets from the child tasks. Now that all data that passes between parent and children is performed with containers, only two **DELETE CHANNEL** commands are required to clean up two types of stored data (Example 6-5):

- Search request information that was passed from parent to child.
- Response information that was passed from child to parent.

*Example 6-5 Housekeeping in parent task*

```
************************************************************************ 01070292
* Issue DELETE CHANNEL for response CONTAINERS.                        * 01070392
************************************************************************ 01070292
DC_0010  DS   0H                                                         00973499
         ST    R14,DC_REG             Save return register               01070893
*                                                                        00806542
         EXEC CICS DELETE                                               X00806642
               CHANNEL(TT_CHAN)                                         X00806764
               NOHANDLE                                                  00806986
*                                                                        00806542
         L     R14,DC_REG             Load return register               01070893
         BCR   B'1111',R14            Return to caller                   01070893
*                                                                        00948799
************************************************************************ 01070292
* Issue DELETE CHANNEL for request  CONTAINERS.                        * 01070392
************************************************************************ 01070292
DC_0020  DS   0H                                                         00973499
         ST    R14,DC_REG             Save return register               01070893
*                                                                        00806542
```

```
         EXEC CICS DELETE                                          X00806642
               CHANNEL(PP_CHAN)                                    X00806764
               NOHANDLE                                             00806986
*                                                                   00806542
         L     R14,DC_REG             Load return register          01070893
         BCR   B'1111',R14            Return to caller              01070893
*                                                                   00948799
*                                                                   00948799
*                                                                   00948799
```

The process changes dramatically for abnormal end conditions. The homegrown solution employed a background ICE transaction. This transaction ran periodically to issue various **INQUIRE** commands, apply logic to identify orphaned constructs, and execute **DELETEQ TS** and **FREEMAIN** commands. Any orphaned storage that was found was thereby released.

With the CICS asynchronous API and the exclusive use of channels and containers, all storage is managed and implicitly freed by CICS. The concerns about orphaned storage are removed. The background housekeeping task from the homegrown solution is entirely eliminated, thereby reducing code, resource usage, and complexity. Also, the high-level design view remains unchanged from the previous stage.

## 6.2 Migration

The differences between the homegrown solution and the new solution that uses the CICS asynchronous API have been described in the sections of this chapter. However, the general steps that we took to move to the new solution might not be obvious. So, here's a summary of the changes to the migration process:

- Initiate child tasks with **RUN TRANSID**, not **START TRANSID**.
- Add channel and child token information to the internal control array.
- In child tasks, use **GET/PUT CONTAINER**s, instead of TS queue and **GETMAIN SHARED**.
- Make the following changes in the parent task:
  - Use **FETCH CHILD TIMEOUT**, instead of the **READQ TS** and **STIMERM**-based polling mechanism
  - Use **GET CONTAINER**, instead of **GETMAIN SHARED**.
  - Use **DELETE CHANNEL**, instead of **DELETEQ TS** and **FREEMAIN** for housekeeping.
- Remove the background process for further housekeeping.

This list demonstrates that the changes needed to convert the homegrown solution to using the CICS asynchronous API are relatively minor. Most of the service remains unchanged.

## 6.3 Summary

Replacing the homegrown asynchronous mechanisms with the CICS asynchronous API required relatively minor changes, but provided significant benefit. The solution was greatly simplified, while maintaining the throughput performance from the original design.

The simplification comes from the removal of numerous items, including these:

- Management of TS queues and shared **GETMAIN**s
- **STIMERM**-based polling mechanism
- Many housekeeping functions

Hundreds of lines of code were removed from the original solution during the conversion.

Flexibility is another benefit with this solution. For example, removal of the STIMERM-based (or other common techniques, like Event Control Blocks or Compare and Swap instructions) polling mechanism eliminates assembly language as a requirement. This change adds options for future maintenance. Also, the success of this change suggests the viability of using higher level languages to develop asynchronous solutions.

Additional expected benefits of the CICS asynchronous API include these:

- Quicker and lower-risk development cycles for similar projects. Some of the "heroics" of the engineering team now might not be needed, because the API abstracts away much of the low-level mechanics that used to be required for these types of solutions.
- Reduced burden of ongoing maintenance of code. The API leads to simpler and easier-to-understand code that can be maintained by developers who are less skilled.
- Greater focus on business value. IBM-supported mechanisms replace proprietary code. This change allows teams to direct more attention to processes that directly add value.

The success of using the CICS asynchronous API in this situation led to plans for using it in other solutions. Chapter 7, “Other implementation patterns” on page 49 shows some of these other design patterns where the API is applicable.

7

# Other implementation patterns

In addition to the implementation that is described in this publication, Walmart leverages the CICS asynchronous API in several other ways. It is not feasible to go into the same level of detail for each of these scenarios. At the same time, it is important to describe the associated architectural patterns that can make extensive use of the capability.

## 7.1 Unordered Responses

The unordered response scenario is related to the EPS search service that has been the main focus of this publication. After the initial requirement had been satisfied, Walmart identified this additional search pattern in a new use case.

Nearly all of the same conditions from the original EPS search service exist in this new situation. The same volume, event structure, data repository, SLAs, and so on, are in place. So, the parallelized I/O solution is also still used. As a reminder, the high-level design of this solution is illustrated in Figure 7-1.

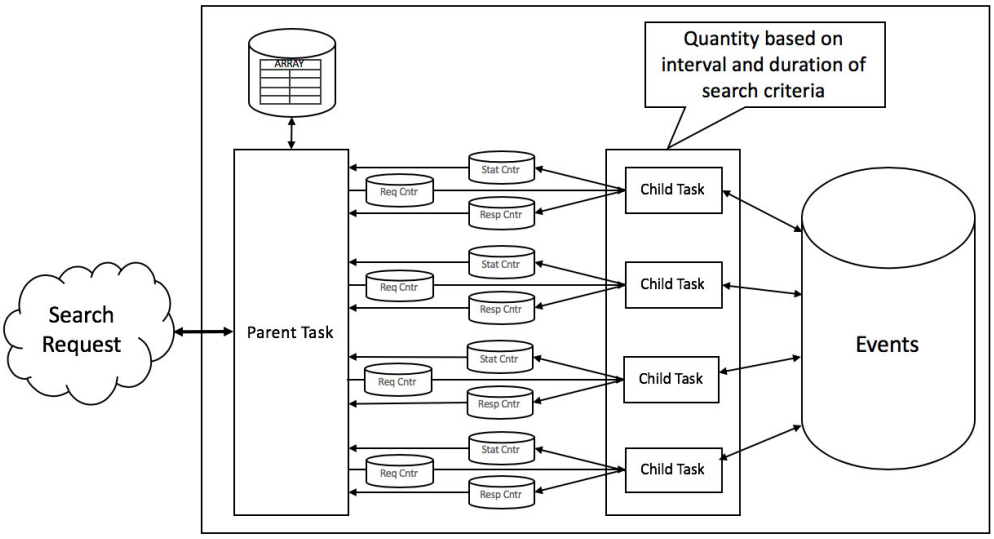



*Figure 7-1 Asynchronous Parallel I/O*

So, this is effectively the same solution. However, there is one key distinction between the original and new use cases. The new use case does not require search results to be returned in chronological order. Consider this example in the new use case: The parent task initiates three child tasks in order, one by one. Unlike the EPS search service scenario in earlier chapters, the responses of the child tasks 1, 2, and 3 are unordered. Child 3, for example, can respond as soon as it finishes with its task, whether or not Child 1's task is complete.

This is a subtle, but important difference that is demonstrated in Figure 7-2.

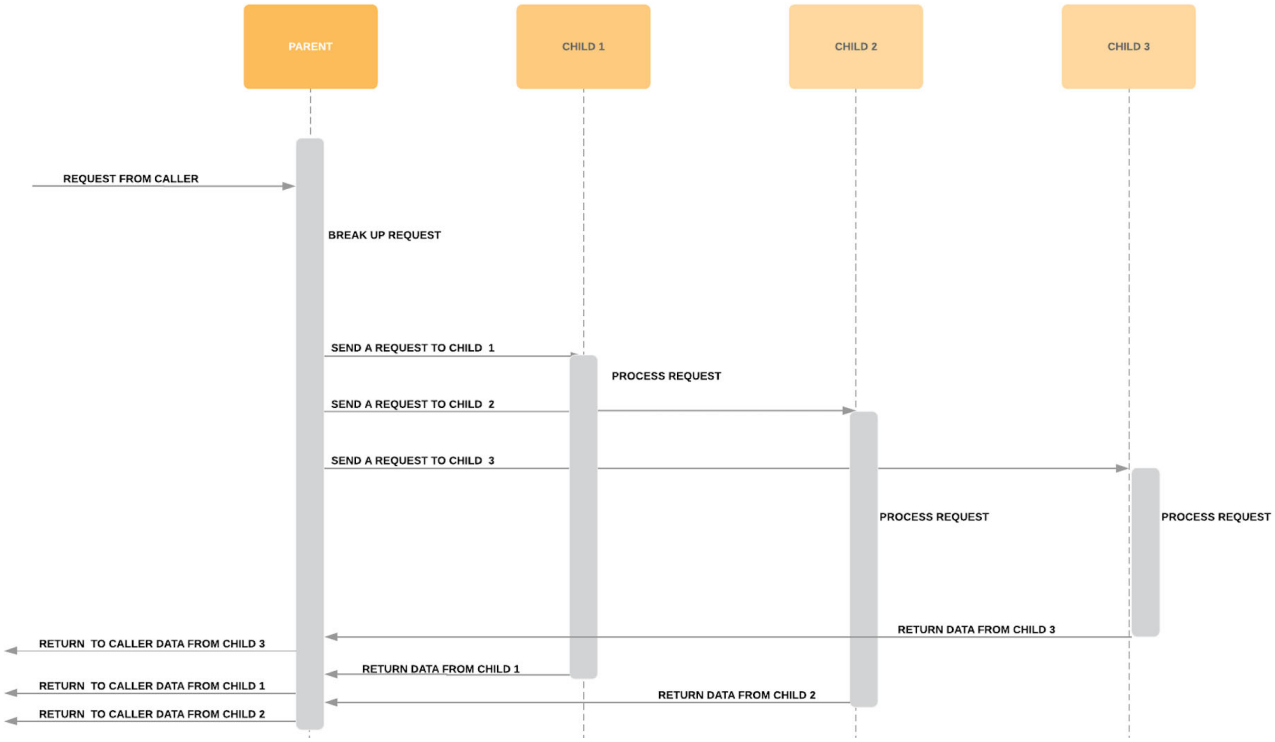



*Figure 7-2 Unordered responses*

Figure 7-2 shows an example of the unordered response scenario. Parent requests move from left to right along the x-axis. Child responses move from right to left. The passage of time is visible from top to bottom in the y-axis.

The sequence of events in Figure 7-2 shows that Child 3 has a faster processing time than the other two children. There can be many reasons for this. Perhaps Child 3 has less data to process or that the data was easier to access. For whatever reason, Child 3 was able to return its data more quickly than the other children, even though it was the last child to be initiated. Eventually, Child 1 responds, then Child 2 responds. The parent task is free from the constraint of returning results to the client in chronological order. As a result, the parent task can fetch and deliver results as they are returned from the children. The **FETCH ANY** command is used to accommodate this behavior.

As with the main use-case covered in Chapter 6, the parent task uses the **RUN TRANSID** command to initiate the child tasks (Example 7-1).

*Example 7-1 RUN TRANSID in parent task*

```
*************************************************************** 00236000
* Issue RUN   TRANSID for child task to check 7 event logs.     * 00237000
* Store Child token in current array entry.                     * 00237000
*************************************************************** 00238000
 1000-RUN-TRANSID.                                              00239000
     ADD  1      TO RC-REQUEST-COUNT.

     EXEC CICS PUT
          CONTAINER(RC-REQUEST-CONTAINER)
          CHANNEL  (RC-REQUEST-CHANNEL)                         02020300
          FROM     (RC-REQUEST)
          FLENGTH  (RC-REQUEST-LENGTH)
          NOHANDLE
     END-EXEC.

     EXEC CICS RUN                                              02020100
          TRANSID (RC-TRANSID)                                  02020200
          CHANNEL (RC-REQUEST-CHANNEL)                          02020300
          CHILD   (RC-CHILD)                                    02020300
          NOHANDLE                                              02020400
     END-EXEC.                                                  02020500

     MOVE RC-CHILD TO RC-CHILD-TOKEN (ARRAY-INDEX).
     MOVE 'A'      TO RC-ENTRY-STATUS(ARRAY-INDEX).

 1000-EXIT.                                                     00239000
     EXIT.
                                                                00235000
```

Then, the **FETCH ANY** command replaces the **FETCH CHILD** command to process the results (Example 7-2):

*Example 7-2 FETCH ANY in parent task*

```
*************************************************************** 00236000
* Issue FETCH ANY for child tasks.                              * 00237000
* Process until every child is complete or time-out occurs.     * 00237000
*************************************************************** 00238000
 2000-FETCH-ANY.                                                00239000
```

```
        MOVE TIME-OUT-VALUE    TO RC-FETCH-TIMEOUT.

        EXEC CICS FETCH                                          02020100
             ANY       (RC-FETCH-CHILD)                          02020200
             CHANNEL   (RC-FETCH-CHANNEL)                        02020300
             COMPSTATUS(RC-FETCH-COMP)                           02020300
             TIMEOUT   (RC-FETCH-TIMEOUT)                        02020300
             NOHANDLE                                            02020400
        END-EXEC.                                                02020500

        IF  EIBRESP       NOT EQUAL DFHRESP (NORMAL)
        OR  RC-FETCH-COMP NOT EQUAL DFHVALUE(NORMAL)
            PERFORM 9100-SEARCH-FAILED  THRU 9100-EXIT.

        PERFORM 2100-CHECK-TOKEN        THRU 2100-EXIT
            WITH TEST AFTER
            VARYING ARRAY-INDEX FROM 1 BY 1
            UNTIL RC-FETCH-CHILD EQUAL RC-CHILD-TOKEN(ARRAY-INDEX).

    2000-EXIT.                                                   00239000
        EXIT.
```

With this approach, the parent still maintains an inventory of the child tasks, but it no longer has to process the results in a particular sequence. It can simply issue the **FETCH ANY** and process any available results until all child tasks have reached completion. This approach typically results in improved responsiveness for the client.

Although this example is specifically related to the EPS service, this general pattern is quite common in asynchronous processing and might be applicable in a great number of situations.

## 7.2 Updating remote hosts (High Latency)

This example somewhat resembles the scenario that is described in 7.1, “Unordered Responses” on page 49. The difference is instead of child tasks interacting with a local data store, they must send information to other remote systems over the network via HTTP.

This implementation is related to the existing zFAM product that was developed by Walmart. The zFAM product is an online object store that functions as a NoSQL (Not only SQL) database.

> Additional information about zFAM is available through the GitHub link and the other IBM Redbook publications, which are described in 2.3.1, “Walmart at a Glance” on page 10.

The product stores and retrieves objects in a local data store as requested from web-based clients. To extend its High Availability (HA) posture, the product must also replicate new or updated objects to multiple other geographically dispersed hosting sites.

The replication activity is expected to be asynchronous and non-blocking to the client. So, this function is assigned to subordinate tasks while the primary task continues on processing the client request. The general depiction of this design is included in Figure 7-3.

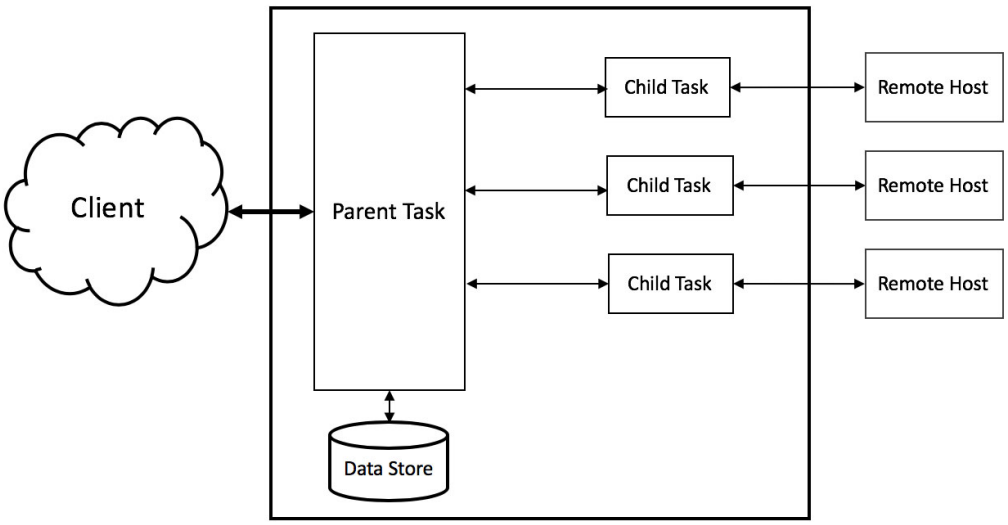



*Figure 7-3 Replicate to remote hosts*

In this example, the order that the child tasks reach completion is not important. But confirmation that they each task actually reaches completion is critical. Figure 7-4 represents the associated process for the parent task:

1. Process the request from the client.
2. Start the child tasks to perform the replication.
3. Provide a response to the client.
4. Note the responses from the child tasks when they complete.

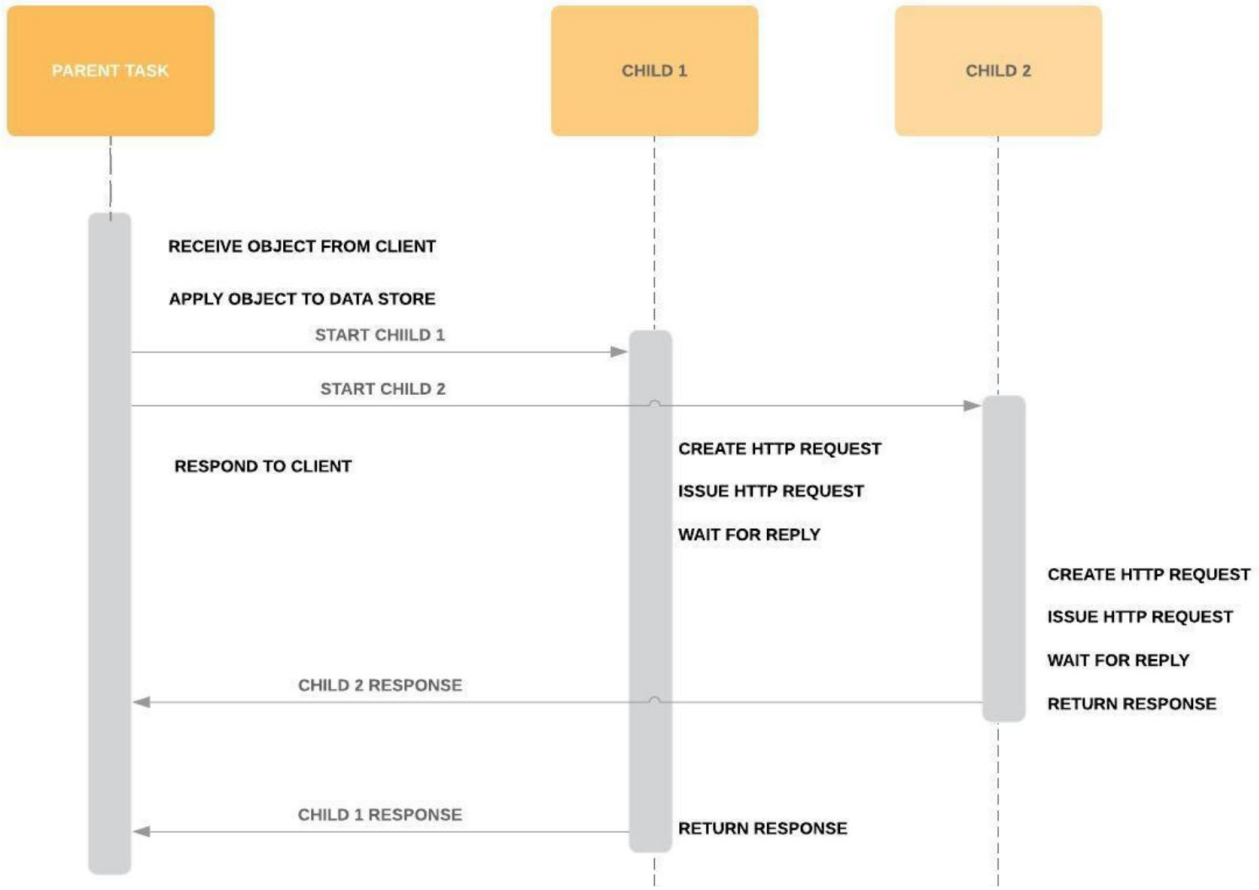



*Figure 7-4 Remote host updates*

This process can be updated to take advantage of the simplicity provided by the CICS asynchronous API. So, the children tasks will be invoked with the **RUN TRANSID** command. Like the example in section 7.1, there is no requirement for ordered responses. So, the **FETCH ANY** command is used to retrieve the completion status from the child tasks. However in this scenario, there is no data to return from the child tasks. Also, the client request has likely been satisfied before the child tasks have completed, so there is no data container to retrieve in this example (Example 7-3 on page 54):

*Example 7-3 FETCH ANY in parent task*

```
*****************************************************************
* Issue FETCH ANY for child tasks.                              *
* Process until every child is complete or time-out occurs.     *
* Channel is not required, as a GET CONTAINER is not issued     *
* for the Child tasks in this process.                          *
*****************************************************************
 2000-FETCH-ANY.
     MOVE TIME-OUT-VALUE    TO RC-FETCH-TIMEOUT.

     EXEC CICS FETCH
          ANY       (RC-FETCH-CHILD)
          COMPSTATUS(RC-FETCH-COMP)
          TIMEOUT   (RC-FETCH-TIMEOUT)
          NOHANDLE
```

```
        END-EXEC.                                                   01640000
                                                                    01650021
        IF  EIBRESP       NOT EQUAL DFHRESP (NORMAL)                01660021
        OR  RC-FETCH-COMP NOT EQUAL DFHVALUE(NORMAL)                01670021
            PERFORM 9100-SEARCH-FAILED  THRU 9100-EXIT.             01680021
                                                                    01690021
        PERFORM 2100-CHECK-TOKEN        THRU 2100-EXIT              01700021
            WITH TEST AFTER                                         01710021
            VARYING ARRAY-INDEX FROM 1 BY 1                         01720021
            UNTIL RC-FETCH-CHILD EQUAL RC-CHILD-TOKEN(ARRAY-INDEX).  01730021
                                                                    01740021
    2000-EXIT.                                                      01750000
        EXIT.                                                       01760021
                                                                    01770000
```

**Note:** Example 7-3 on page 54 is similar to previous examples of the **FETCH ANY** command. However, in this scenario the **CHANNEL** parameter has been omitted as no container data is expected back from the child transcation

Some of the details in this example are particular to Walmart's specific use-case. Nonetheless, this general pattern is seen frequently in applications. Any time that relatively high-latency communications across the network are required, asynchronous processing has value. For example, if you are using REST (Representational State Transfer) interfaces or SOAP (Simple Object Access Protocol) services, it is typically beneficial to process those communications asynchronously. A key benefit is that you avoid blocking the primary task from doing other work. The CICS asynchronous API helps to simplify this processing activity.

## 7.3 Fire-and-Forget

The previous examples demonstrate the use of asynchronous processing that is focused on forms of parallel or concurrent processing. In contrast, the pattern in this section involves a Fire-and-Forget model. In this example, we do not use a child task to gather data that the parent needs. Instead, we focus on initiating children that act autonomously without returning data, or even status information, to the parent.

The previous examples in this chapter were also primarily used to highlight the simplicity that the IBM CICS asynchronous API provides. This section shifts that focus to other important dimensions: efficiency and performance.

This scenario involves a system service that performs this process:

1. Capture information assets (for example, purchase orders) that flow through one application.
2. Sending them into another application for other processing.
3. As the assets are captured, initiate a child task that is associated with each asset and interacts with the receiving application.

As previously noted, these child tasks act autonomously and are never fetched by the parent task. Figure 7-5 shows a very simple view of this process.

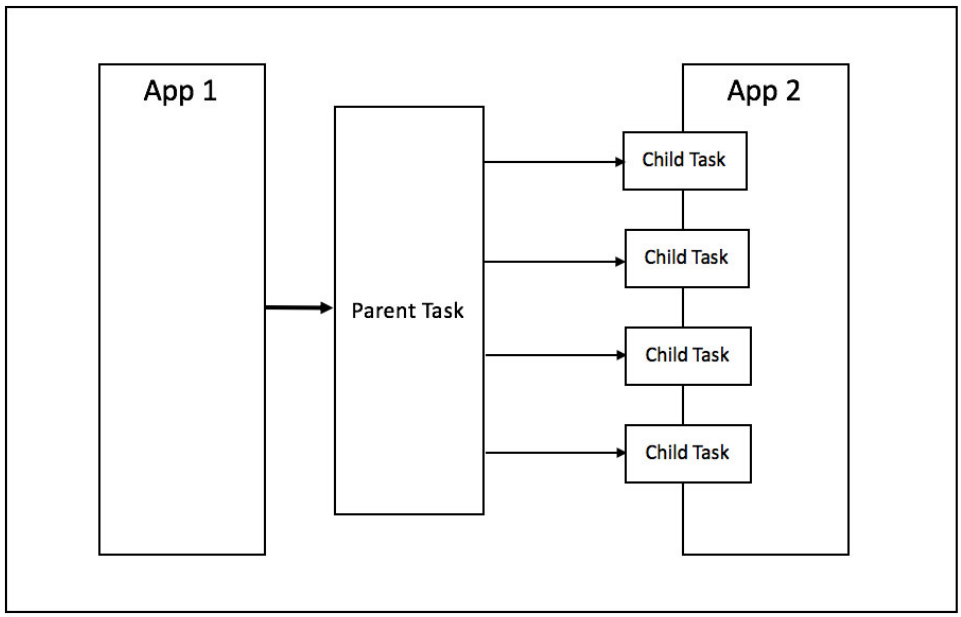



*Figure 7-5 Fire-and-Forget*

In reality, a Fire-and-Forget service as described here can be very complex and well beyond the scope of this publication. The description of the process is provided only to give some basic context for the scenario. The important part is that, up to now, we used the **START TRANSID** command to initiate the child tasks. Now, this function is being replaced with the **RUN TRANSID**. As a result, we realize significant performance improvements.

It should first be noted that simplicity is still a relevant characteristic in this example. In particular, the change from **START TRANSID** to **RUN TRANSID** is a relatively easy and unintrusive modification. For example, here is the code to issue the **START TRANSID** (Example 7-4):

*Example 7-4 Fire and Forget START TRANSID in parent task*

```
*****************************************************************
* Start - Data for child task                                   *
*****************************************************************
 01  SC-CHANNEL            PIC  X(16) VALUE 'CHILD-CHANNEL   '.
 01  SC-CONTAINER          PIC  X(16) VALUE 'CHILD-CONTAINER '.
 01  SC-TRANSID            PIC  X(04) VALUE 'RICH'.
 01  SC-LENGTH             PIC S9(08) COMP VALUE ZEROES.

 01  SC-REQUEST.
     02  SC-PO-NUMBER      PIC  9(10) COMP-3 VALUE ZEROES.
     02  SC-PO-SEQUENCE    PIC  9(08) COMP   VALUE ZEROES.
     02  SC-PO-SEGMENT     PIC  9(02) COMP   VALUE ZEROES.
     02  SC-PO-SUFFIX      PIC  9(08) COMP   VALUE ZEROES.

*****************************************************************
* End   - Data for child task                                   *
*****************************************************************
*****************************************************************
*****************************************************************
* Issue START TRANSID for child task to process PO numbers.     *
*****************************************************************
 1000-START-TRANSID.
```

```
    MOVE LENGTH OF SC-REQUEST  TO SC-LENGTH.

    EXEC CICS PUT
         CONTAINER(SC-CONTAINER)
         CHANNEL  (SC-CHANNEL)                                  02020300
         FROM     (SC-REQUEST)
         FLENGTH  (SC-LENGTH)
         NOHANDLE
    END-EXEC.

    EXEC CICS START                                             02020100
         TRANSID (SC-TRANSID)                                   02020200
         CHANNEL (SC-CHANNEL)                                   02020300
         NOHANDLE                                               02020400
    END-EXEC.                                                   02020500

 1000-EXIT.                                                     00239000
    EXIT.
                                                                00235000
```

Notice the similarity of the **RUN TRANSID** code that replaces the preceding code (Example 7-5):

*Example 7-5 Fire and Forget RUN TRANSID in parent task*

```
*****************************************************************00236000
* Issue RUN   TRANSID for child task to process PO numbers.     * 00237000
*****************************************************************00238000
 1000-RUN-TRANSID.                                              00239000
    MOVE LENGTH OF SC-REQUEST  TO SC-LENGTH.

    EXEC CICS PUT
         CONTAINER(SC-CONTAINER)
         CHANNEL  (SC-CHANNEL)                                  02020300
         FROM     (SC-REQUEST)
         FLENGTH  (SC-LENGTH)
         NOHANDLE
    END-EXEC.


    EXEC CICS RUN                                               02020100
         TRANSID (SC-TRANSID)                                   02020200
         CHANNEL (SC-CHANNEL)                                   02020300
         CHILD   (SC-CHILD)                                     02020300
         NOHANDLE                                               02020400
    END-EXEC.                                                   02020500

 1000-EXIT.                                                     00239000
    EXIT.
                                                                00235000
```

Now, compare the effect of these innocuous code adjustments on performance metrics for the process. The baseline metrics with using the **START** command are as follows (Example 7-6):

*Example 7-6 Fire and Forget Performance START TRANSID in parent task*

```
15FEB2019  17:56:12 ------ MAINVIEW WINDOW INTERFACE (V6.3.00) ----------------
 COMMAND  ===>                                                 SCROLL ===> CSR
 CURR WIN ===> 1        ALT WIN ===>
 >W1 =CHIST============CICS001==*========15FEB2019==17:56:12====MVCICS===D===88
  Task    CICS   Lcl Task   Lcl Task Tran Response    CPU    Storage  I/O  Term
  Num    System  End Date   End Time  ID    Time      Time     HWM   Calls  ID
  45010 CICS001  15FEB2019  17:55:24 JOHN   0.00303  0.00062   58976     0 ZAAA
  45020 CICS001  15FEB2019  17:55:24 RICH   0.00106  0.00008   55392     1
  45019 CICS001  15FEB2019  17:55:24 RICH   0.00097  0.00008   55392     1
  45017 CICS001  15FEB2019  17:55:24 RICH   0.00113  0.00010   55392     1
  45018 CICS001  15FEB2019  17:55:24 RICH   0.00091  0.00009   55392     1
  45016 CICS001  15FEB2019  17:55:24 RICH   0.00107  0.00009   55392     1
  45013 CICS001  15FEB2019  17:55:24 RICH   0.00132  0.00009   55392     1
  45014 CICS001  15FEB2019  17:55:24 RICH   0.00128  0.00009   55392     1
  45015 CICS001  15FEB2019  17:55:24 RICH   0.00062  0.00009   55392     1
  45012 CICS001  15FEB2019  17:55:24 RICH   0.00080  0.00012   55392     1
  45011 CICS001  15FEB2019  17:55:24 RICH   0.00059  0.00020   55392     1
```

The metrics that are associated with using the **RUN TRANSID** command are as follows (Example 7-7):

*Example 7-7 Fire and Forget Performance RUN TRANSID in parent task*

```
15FEB2019  17:56:12 ------ MAINVIEW WINDOW INTERFACE (V6.3.00) ----------------
 COMMAND  ===>                                                 SCROLL ===> CSR
 CURR WIN ===> 1        ALT WIN ===>
 >W1 =CHIST============CICS001==*========15FEB2019==17:56:12====MVCICS===D===88
  Task    CICS   Lcl Task   Lcl Task Tran Response    CPU    Storage  I/O  Term
  Num    System  End Date   End Time  ID    Time      Time     HWM   Calls  ID
  44941 CICS001  15FEB2019  17:52:35 JOHN   0.00108  0.00054   58976     0 ZAAA
  44939 CICS001  15FEB2019  17:52:33 RICH   0.00144  0.00009   55392     1
  44940 CICS001  15FEB2019  17:52:33 RICH   0.00140  0.00008   55392     1
  44934 CICS001  15FEB2019  17:52:33 RICH   0.00145  0.00008   55392     1
  44937 CICS001  15FEB2019  17:52:33 RICH   0.00136  0.00008   55392     1
  44938 CICS001  15FEB2019  17:52:33 RICH   0.00134  0.00009   55392     1
  44936 CICS001  15FEB2019  17:52:33 RICH   0.00134  0.00009   55392     1
  44935 CICS001  15FEB2019  17:52:33 RICH   0.00114  0.00008   55392     1
  44931 CICS001  15FEB2019  17:52:33 RICH   0.00117  0.00015   55392     1
  44933 CICS001  15FEB2019  17:52:33 RICH   0.00092  0.00009   55392     1
  44932 CICS001  15FEB2019  17:52:33 RICH   0.00079  0.00012   55392     1
```

In comparing these metrics, you see significant differences in the parent task (Tran ID = `JOHN`).

*Table 7-1 Comparative metrics for the START and RUN TRANSID commands*

| | **START command** | **RUN TRANSID command** |
|---|---|---|
| **Response Time** | 0.00303 (3.03 ms) | 0.00108 (1.08 ms) |
| **CPU Time** | 0.00062 (.62 ms) | 0.00054 (.54 ms) |

These values significantly improve performance for the parent task. A 13% reduction in CPU consumption is observed, which is enough to be beneficial. More importantly, we see a 65% reduction in response time. These improvements arise because the **RUN TRANSID** command is threadsafe. Thus, much of the task switching of the non-threadsafe **START** command is eliminated. You can verify this improvement by inspecting the **AUXTRACE** of the tasks.

At scale, with large numbers of transactions, these improvements have a substantial positive impact on system throughput. Any pattern that involves a task that starts groups of subordinate tasks, you should consider a change to the **RUN TRANSID** command. This humble, low-investment change can dramatically improve performance with no additional, or even slightly lower CPU consumption.

## 7.4 Summary

This chapter looked into a few other examples where the Walmart team is leveraging the CICS asynchronous API to improve processing. These examples also represent implementation patterns that are common in many applications and that might benefit from employing the CICS asynchronous API. Potential benefits include ease-of-use, simplification, and better performance. However, there are always issues to consider when you adopt new technology. Chapter 8, “Considerations” on page 61 will highlight some of these issues.

8

# Considerations

Throughout this Redbooks publication, we have followed the journey of Walmart as they adopted the IBM CICS asynchronous API within an offering that is heavily used in their organization. As we have read in earlier chapters, this is not their first encounter with asynchronous processing patterns in CICS applications, and they have experience with creating their own frameworks. They have a wealth of experiences to share with readers who might also be on the journey of adopting asynchronous patterns in their applications.

In this chapter, we share advice from the Walmart team, including some of their suggestions of practices, things to be aware of, and how they approached challenges.

It is likely that readers who are implementing asynchronous patterns will encounter similar challenges. Their specific nature will likely depend on the specific environments and requirements. Although further investigations will likely be required, this chapter should provide valuable food for thought.

Some advice in this section is equally applicable to asynchronous solutions that do, and do not take advantage of the CICS asynchronous API.

## 8.1 Transactionality and recovery

Use of the CICS asynchronous API greatly simplifies the efforts of creating an asynchronous processing pattern in enterprise applications. However, application developers are still required to understand the transactionality of the parent and child transactions. This is especially true when you are considering the impact of invocations that have failed midway.

Take for example an application that **`PROGRAM LINK`**s to two programs serially. If the second program ABENDs, the application can be automatically rolled back by CICS, with no additional developer foresight.

Now, consider a similar application that calls an additional two programs asynchronously. If the second of those programs ABENDs, what is the impact? The parent and two child processes are full CICS transactions in their own rights. Hence, the ABEND of second child, does not imply that the first child will be rolled back. In fact, the first child might still be

executing, it might have terminated and committed; indeed, it might not have started yet. There is also a possibility that the parent has completed, or even ABENDed.

It can be safer to use only **GET** operations for child transactions, for example, asynchronous retrieval for stock quotes, or addresses. If something goes wrong, it is safe to simply throw away the results. However, if a child changes the state of the system (such as Child 1 '**minus £200 from the current account**' and Child 2 '**add £200 to the savings account**'), the developer must account for partial failures. Rollback algorithms and compensation flows are typically required for **POST** operations.

These Unit of Work (UOW) scenarios can be a source of difficulties for application developers who are new to asynchronous patterns. The CICS asynchronous API helps greatly with these combinations:

- The management of resources during transaction termination
- Saving/cleaning state

Nonetheless, the application developer needs education and skills to keep track of the interplay between parent and child transactions.

## 8.2 Data integrity

When data is passed between parent and child transactions, the data itself (or a copy of the data) should be passed, not a reference to the data. This practice ensures that each party has a sound view of the data, and that it will not be changed accidentally by another entity.

The CICS asynchronous API achieves this goal by using CICS channels and containers. The **RUN TRANSID** command can optionally specify a channel to pass data to a child transaction. The command gives a copy of the data to the child. Hence the parent and child have their own view of the data, and data integrity can be maintained. On a **FETCH CHILD** or **FETCH ANY** command, a channel can be returned. The returned channel is renamed by the API, so that it does not clash with an existing channel that the parent owns. It is important to remember to use the new name for the channel that the child returns!

## 8.3 Timeouts

You can issue the **FETCH CHILD** and **FETCH ANY** API commands with an optional **TIMEOUT** parameter. It is suggested to always code the **TIMEOUT** parameter and to supply the field with a variable (perhaps read in the values from some properties file). By doing this, you have future durability of code. You can always change the timeout value (or specify `0` which indicates that the timeout should be ignored), without the need to alter the source code.

## 8.4 CPU / Cost

The CICS asynchronous API solution might affect the amount of resources that the system consumes. Understanding the impact of design choices helps with the overall success of the application.

In this publication, we've included four examples of using the Asynchronous API. Three of those examples use request and response information that is shared between the parent and child tasks. The fourth example is a 'Fire-and-Forget' design and is covered in 7.3,

“Fire-and-Forget” on page 55. In that example, the parent provides information to the child task, but does not check for the completion, order, or response information regarding the child tasks. Using this model and changing from non-threadsafe **START TRANSID** command to the threadsafe **RUN TRANSID** shows a reduction in CPU. This model also delivers an increase in throughput by reducing each child task's response time.

### 8.4.1 Command overhead

A performance study was conducted to determine the difference in initiating a child task with **EXEC CICS RUN TRANSID** versus by using the **EXEC CICS START** command. The test and its detailed results are documented in *IBM CICS Performance Series: CICS TS for z/OS V5 Performance Report,* SG24-8298. The performance study found that the **EXEC CICS RUN TRANSID** is no more expensive to execute than the simpler **EXEC CICS START** command. Yet, it provides all the benefits of the CICS asynchronous API. Here is a summary of those findings:

> Using the asynchronous service of the EXEC CICS RUN TRANSID command to initiate child tasks has approximately the same CPU cost as using the EXEC CICS START command. There is a small amount of overhead for EXEC CICS START due to the amount of TCB change mode processing (because the command is not threadsafe). Other ways exist to manage communication between parent and child tasks, but none are as efficient as using the asynchronous API. Practices such as the polling of CICS services, use of intrapartition transient data trigger queues, and external resource managers like IBM MQ, all require additional CPU processing. Therefore, they increase the management and monitoring overhead.

These assertions are further supported by the assessments that Walmart made, which are discussed in 7.3, “Fire-and-Forget” on page 55.

### 8.4.2 Justification for additional processing

Chapter 4, “Our initial sequential approach” on page 19 examined the initial approach that Walmart took to determine whether sequential processing would be sufficient to accommodate the requirements of a particular application. In that case, it was determined that sequential processing was not sufficient and that a framework for parallelism that uses asynchronous methods would be necessary.

Walmart's Event Processing System (EPS) search service was initially developed for that one application. But it is a service that could also be deployed for and leveraged by other applications that might not have as extreme requirements.

Despite the benefits provided by the CICS asynchronous API, there is still a considerable amount of overhead for coordination, collation, and management, which ensure asynchrony and parallelism for that service. So, the service was built to also be deployable as a sequential solution for applications with less demanding requirements. That way, EPS does not incur extra processing unnecessarily.

This concern might not apply to all implementations of asynchronous processing. Each potential design must be assessed individually. However, it's worth noting that additional complexity and overhead might not be warranted, even in deployments of a single service with differing circumstances.

## 8.5 Resources

Other system resources and settings are relevant when you consider asynchronous implementations. These resources and settings can affect not only the CICS regions that are involved, but the entire system. Some prominent examples are highlighted in this section.

### 8.5.1 Threadsafe considerations

IBM CICS asynchronous API commands are threadsafe and do not need to switch between TCBs. This feature saves CPU time and lowers response time in your application. The MAXOPENTCB parameter in the System Initialization Table (SIT) controls the number of TCBs that are allocated in a CICS region. You use the **`CEMT INQUIRE MAXOPENTCB`** command to access and manage this parameter. Your setting for this parameter has either a positive or negative effect on the dispatching and performance of both the parent and child tasks within a region.

The OMVS parameters MAXPROCSYS and MAXPROCUSER also affect multi-threaded tasks within a system and within a region. You must review these settings and possibly adjust them to support the number of OTE TCBs that are used by the increased number of concurrent parent and child tasks.

### 8.5.2 Managing MXT and TRANCLASS

The number of tasks that are allowed to run in the system affects system performance. If the maximum task specification (MXT) is set too high, tasks might not get resources in a timely fashion. If the maximum task specification (MXT) is too low, tasks might be delayed because they are waiting to be dispatched. The IBM Redbooks publication *IBM CICS Asynchronous API: Concurrent Processing Made Simple*, SG24-8411 provides a detailed discussion and formula for calculating MXT.

Along with setting MXT, attention should be given to MAXOPENTCB. As discussed in the previous topic regarding threadsafe considerations, these values affect each other.

The TRANCLASS parameter is another resource that affects the dispatching of both the parent and child tasks. In Walmart's EPS design, which is described in Chapter 2, Walmart assigned a TRANCLASS for the parent task and a different TRANCLASS for the child task. The maximum child tasks that the parent task initiates in EPS can reach 255. In turn, the value of TRANCLASS for the child task is set to a value considerably higher than the value for the parent task. The separation of parent and child tasks into different TRANCLASS names is critically important to a successful parallel process.

## 8.6 Testing

Testing of asynchronous patterns can be tricky. There are many combinations of states for the parent and child transactions. Also, it might be difficult to force timing windows, and to be sure that niche timing windows have been tested appropriately. For example, do you know what will happen if Child 1 is delayed, Child 2 is queued on a transaction class, and the parent has terminated already?

Testing of the Walmart EPS service was performed by using two techniques. The EPS parent task is invoked by a REST request. So, both of the techniques involved some form of a REST

client. The initial testing was performed through a REST client, of which there are several on the market for little or no cost.

More advanced load testing with multiple client requests was performed for stress test analysis, resource consumption, and contention identification.

**Hint for developer testing with multiple child transactions:**
It might be beneficial to give each child a different transaction ID, even though they might eventually have the same ID. With this practice, you maintain control over individual child transactions, easily identify them, and use techniques such as CEDF/CEDX to force specific scenarios.

### 8.6.1 Testing that uses a single REST client

Walmart used the following process to test with a single REST client:

1. Perform diagnostics on the parent task with CEDX by logging on to the target CICS region.
2. On a different 3270 device, log on to the same CICS region and set CEDX for a child task.
3. Control the flow of information between the tasks by running CEDF on the parent and CEDX on the child.
4. Hold the child task with a timeout request to gain visibility into error handling and the clean-up process.

### 8.6.2 Testing that uses multiple REST clients

More advanced testing was performed. A driver transaction was created in a CICS region that issued several concurrent 'Fire-and-Forget' child tasks. The tasks used either the **START TRANSID** or **RUN TRANSID**, where the child task was a REST client that issues **WEB SEND** and other WEB-related API commands. Each of these child tasks would create an HTTP REST request that specifies the URL of the EPS task. This EPS task can be processed by a single CICS region. Or the task can be load balanced across multiple CICS regions by using the z/OS TCP Sysplex Distributor and WLM.

Initial testing of the driver transaction was performed by using CEDF to ensure that initiation of the child tasks was performed properly. CEDX was also used in the same region to capture a REST client child task. This approach ensured a successful invocation of the EPS service through use of the WEB SEND command. The test team used the same method of capturing CEDF of the parent task and CEDX of a child task in the target EPS region. This approach ensured that the initial testing that used multiple REST clients was successful.

When this initial testing proved satisfactory, it was time for the next steps:

1. Use the driver transaction, child client tasks, parent EPS, and child EPS tasks without any diagnostics or restriction to the work flow.
2. At this point, use CICS monitoring software to review the region and transaction, and perform and analyze the impact on resources, such as MXT, MAXOPENTCB, TRANCLASS, DSA and EDSA, and others.
3. Finally, use SMF and IBM RMF™ to evaluate the impact on the CICS region and on z/OS resources when different volumes of increased EPS parent and child tasks run.

## 8.7 Skills

*Keep it simple … seriously!*

For all the benefits of asynchronous patterns, they do imply a higher cognitive load on viewers of the algorithm. This load affects the initial application developers, and also the defect fixers, future feature developers, servicing teams, and anyone that clones the code for use elsewhere.

As we discussed in 2.1, “Asynchronous Processing” on page 6, many clients have used different technologies to implement their own asynchronous frameworks. Typically, they have many parts, including **START**s, polling **DELAY**s, ECBs, TS queues, clean up transaction, and so on. Each technology brings with it a set of complexities.

The CICS asynchronous API is IBM-supported and is designed with simplicity in mind. Our advice is to keep your logic as simple as possible. For example, you might not need to issue a **FREE CHILD** command if your application is just about to complete.

## 8.8 Conclusion

Asynchronous processing can be greatly beneficial to increased performance and improved user experiences for your applications. However, it can also be complicated and difficult to implement. In this publication, we highlighted all of these concepts with real-world situations that the Walmart engineering team experienced. We also strove to demonstrate that the introduction of the CICS asynchronous API took z/OS developers significantly farther toward the realization of these benefits, while reducing the associated difficulties. The CICS asynchronous API brings simplicity, performance, and stability into the asynchronous processing realm. We hope that it inspires creativity and empowers all readers to build better services for our businesses.

Redbooks

# Walmart and the CICS Asynchronous API:

SG24-8444-00

ISBN DocISBN

IBM

(1.5” spine)
1.5”<-> 1.998”
789 <->1051 pages

Redbooks

Walmart and the CICS Asynchronous API:

SG24-8444-00

ISBN DocISBN

IBM

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Redbooks

Walmart and the CICS Asynchronous API: An Adoption Experience

SG24-8444-00

ISBN DocISBN

IBM

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Redbooks

Walmart and the CICS Asynchronous API: An Adoption Experience

(0.2”spine)
0.17”<->0.473”
90<->249 pages

(0.1”spine)
0.1”<->0.169”
53<->89 pages

Redbooks

# Walmart and the CICS Asynchronous API:

SG24-8444-00

ISBN DocISBN

IBM

(2.5” spine)
2.5”<->nnn.n”
1315<-> nnnn pages

Redbooks

# Walmart and the CICS Asynchronous API: An Adoption Experience

SG24-8444-00

ISBN DocISBN

IBM

(2.0” spine)
2.0” <-> 2.498”
1052 <-> 1314 pages

SG24-8444-00

ISBN 0738457477

Printed in U.S.A.