Redbooks
ibm.com/redbooks

IBM CICS and the Coupling Facility
Beyond the Basics

Arndt Eade

Randy Frerking

Rich Jackson

Kellie Mathis

z Systems

IBM

Redbooks

International Technical Support Organization

IBM CICS and the Coupling Facility: Beyond the Basics

February 2018

SG24-8420-00

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

First Edition (February 2018)

This edition applies to Version 5, Release 4 of IBM CICS Transaction Server.

Contents

Notices

Trademarks

CICS®	OS/390®	VTAM®
DB2®	Parallel Sysplex®	z/OS®
IBM®	Redbooks®	
MVS™	Redbooks (logo) ®	

Preface

It’s easy to look at the title of a book and think “that's old news” or “I already know all there is to know on that subject.” But before you dismiss this publication, consider just how far the IBM® Parallel Sysplex® architecture has come. From the early days the mainframe has embraced a shared everything approach. The original designers coded IBM z/OS® (called IBM *MVS*™ or IBM *OS/390*® back then) with the functionality necessary for the operating system to create the repositories, manage the data flow, and ensure the integrity of the systems involved. From there, the middleware systems provided the exploitation and advanced functions to mature the technology. The component in the middle of all this great technology can easily be taken for granted. That is the IBM Coupling Facility.

This IBM Redbooks® publication discusses both traditional uses for the IBM Coupling Facility technology and new ways to use it with products such as IBM CICS®. You can learn how to perform new functions and have these functions benefit from the scalability and availability achieved only in a mainframe ecosystem.

Open standards are a large part of considerations today, as most companies run IT shops with a mix of technology components. As the world embraces these technologies, it is necessary to understand how to mix the world of mainframe architectures and products with other open architectures. This mix allows the best tool to be used to solve processing needs, at the right cost and service levels.

Often the functions needed for modern processing can be found in house, in places where staff are skilled and that already deliver the robust production environments you count on daily. This book discusses these modern functions and how to achieve them with CICS use of the IBM Coupling Facility. You will learn how one IBM client, Walmart, took these concepts far beyond the original design as they share their experiences and even share code examples to help you get started.

The last chapter of this book shows what can be achieved when a combination of old and new functions are use together. Even if you have familiarity with what could be done with the IBM Coupling Facility in the past, there is much to learn and deploy in a modern world.

Those who are familiar with the IBM Coupling Facility might find the content of this book helpful. Additionally, readers who are considering how to use the IBM Coupling Facility technology within their environment might also find useful information in the chapters that follow.

Authors

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Hursley Center.

Figure 1 Left to right: Rich, Randy, Kellie, Arndt, and Martin

Arndt Eade is a senior software engineer working in the CICS Development Services team at the IBM Hursley laboratory in Hursley, United Kingdom. Arndt has 35 years of IT experience working in operations, application development, and systems programming on zSeries products, such as CICS and IBM MQ. He joined IBM in 2003 and currently runs the CICS Development Services team.

Randy Frerking is a Distinguished Systems Engineer on the 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 38 years of CICS and mainframe middleware experience with 10 years at Walmart Technology and has been selected as an IBM Champion for 2018.

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 has been selected as an IBM Champion for 2018.

Kellie Mathis is an IT executive who has worked in the area of mainframe ecosystem for 33 years. She recently retired from IBM and now leads the mainframe team at Direct Systems Support. Kellie's background includes z/OS, Z systems hardware, Z software, storage, and DR. She regularly consults with clients to help them get the best value from their mainframe environment. Kellie works with many mainframe and non-mainframe executives to educate and advise on modern ways to use technology.

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

Special thanks to **Catherine Moxey**, IBM Senior Technical Staff Member, CICS Analytics and Performance for her guidance and reviews.

Now you can become a published author, too!

Comments welcome

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

Understanding the IBM Coupling Facility

For many environments, having an IBM Coupling Facility (CF) as a part of the infrastructure has become routine. Companies have grown accustomed to planning resources, such as memory, in order to support structures, to plan for connectivity to the z/OS images, and to make the CF the center of the availability strategy for the Parallel Sysplex. Have you ever stopped to ask whether there is more the CF can do for you? Are you taking advantage of everything you can and should be from the CF?

In 1957, what is now Bubble Wrap[1] was invented by Alfred Fielding and Marc Chavannes. Their vision was wallpaper during a time when funky wall coverings were all the rage. Although they weren't successful in their endeavor, a few years later a salesman named Frederick Bowers stumbled onto a new way to use the product. IBM had just launched the IBM 1401 as one of the world's first business computers. Bowers showed IBM how Bubble Wrap could be used to protect the 1401 sensitive parts in transit. And the rest is history! Bubble Wrap is a prime example of where the inventor didn't know how the product would eventually be used after creative and experienced people had a chance to apply their imagination.

This book shows new uses for the IBM CF and discusses methods that a large client is using the CF today, with the goal of inspiring you to add to the growing list of creative capabilities that can be achieved. This chapter provides an overview of the background of the historical use of the CF, the newer functions of the CF, and the benefits of reading this book. It includes the following topics:

- Coupling Facility defined
- Historical uses of Coupling Facility structures
- Newer functions of the CF structures
- Benefits of reading this book
- Client experience
- Moving forward

[1] Bubble Wrap is a generic trademark owned by Sealed Air Corporation

1.1 Coupling Facility defined

The CF was announced as part of the IBM original sysplex announcement in 1994. A *coupling facility* can be defined as a shareable storage medium that makes data sharing possible by assuring integrity and consistency of data. It helps to maintain the availability of the sysplex. The CF allows software on different systems in the sysplex to share data with integrity.

1.1.1 Coupling Facility structures

The CF contains both processing capacity and storage. The storage element has separate containers, called *structures*, and each structure has a specific function in the role of the Parallel Sysplex as follows:

- Cache
- List
- Lock

All three types of structures are used in the functional use cases described further in this book. For a detailed description about how to set up a CF, see *z/OS MVS Setting Up a Sysplex*, SA23-1399.

1.1.2 Coupling Facility placement

You have the following choices for where the CF is located in the Parallel Sysplex:

- A CF can be defined in a logical partition (LPAR) that can be located in a stand-alone IBM Z server that houses only CF LPARs. This implementation provides the best isolation and, therefore, the best availability.
- The other option is to define a CF LPAR on an IBM Z server that also contains z/OS LPARs in the Parallel Sysplex.

Either of these implementations can be used to achieve the functional use cases described further in this book. For information about the pros and cons of each type of implementation, refer to *System z Parallel Sysplex Best Practices*, SG24-7817.

1.2 Historical uses of Coupling Facility structures

The CF was originally designed to provide data sharing across the systems in a sysplex while maintaining the integrity and consistency of shared data. It is the necessary element that is needed to provide availability of a sysplex. Using the CF can be a challenge because it represents a method to connect systems and control data in a shared environment. The following are common uses for CF structures:

- Global resource serialization (GRS) lock
- Cross-system Coupling Facility (XCF) signaling
- Group buffer pools
- Logging
- JES2 checkpoint
- Internal resource lock manager (IRLM) lock
- MQ
- Virtual Telecommunications Access Method (IBM VTAM®) generic resource

For detailed information about each structure, see *z/OS MVS Setting Up a Sysplex*, SA23-1399.

1.3 Newer functions of the CF structures

The CF functionality was later enhanced to provide the following additional benefits for Parallel Sysplex users:

- *Named Counter*: Provides a facility for generating unique sequence numbers for use by application programs. Examples are a unique number for orders or invoices.
- *CF Data Table*: Provides a method of file data sharing without the use of a file owning region and allows rapid sharing of working data within a sysplex.
- *Temp Storage*: Provides a way to store data that must be available to multiple transactions. A good alternative for sharing non-recoverable, temporary storage queues.
- *VSAM RLS*: Provides an access method for VSAM data sets to be shared with full update capability between applications running in many CICS regions and batch.

These methods are used by CICS programs and can greatly enhance the speed and functions that are available across system instances.

Chapters 2 through 5 of this book provide more detail into the setup and use of each of these functions. GRS is one of the original CF uses and is required to provide global serialization for the Parallel Sysplex. Chapter 6 discusses creative uses of GRS structures that might not have been considered in traditional use within application programs.

1.4 Benefits of reading this book

Readers with a basic familiarity of Parallel Sysplex and CICS might benefit from the discussion presented in this book. Specific benefits for each role are listed in the sections that follow. This book is arranged to allow drilling down into the specific CF function by chapter. The advanced-knowledge reader who is already familiar with CF functions might benefit from Chapter 7, “Combined functional use” on page 57, which discusses combined functional use of these functions as they are applied to real business issues.

1.4.1 Design developer benefits

The person who is responsible for the design and development of overall solutions is also concerned with the overall data flow, performance, and resource cost of solutions. By understanding the functions that are available for use in the Coupling Facility, the person in this role can benefit by understanding better the following methods:

- Removal of limitations of a single region or LPAR
- Designs that enable heavy workloads to be handled with availability built in
- Support for multiple channels

1.4.2 Application programmer benefits

The person responsible for application programming is constantly being asked to do more and to do it faster. The person in this role can benefit by understanding better the following methods:

- Gain flexibility in managing resources without having to change the program
- Find available resources to reduce the number of commands to perform a function
- Realize greater availability and scalability from the underlying infrastructure

1.4.3 System programmer benefits

System programmers have a full view of the underlying system architecture and are responsible for implementing the technology to enable higher level functions. The better they do their jobs, the more transparent their work is to their users. The person in this role can benefit by understanding better the following methods:

- Providing high availability that is not limited to a single view of the application
- Implement changes to the operating system, underlying middleware, and so forth with no impact to a user
- Remove scheduled change issues for applications that run only in a single region or a single LPAR or use local VSAM

Applications that make use of the Coupling Facility allow benefits across the board, from design to managing resources to maintaining the infrastructure. — Kellie Mathis, IT Executive, Direct Systems Support

1.5 Client experience

Design in theory is just design until it is implemented and given life. From the early Parallel Sysplex days, Walmart has been a pioneer in using IBM Z server technologies and architectures, such as Parallel Sysplex, and products, such as IBM CICS. The first implementations of Coupling Facilities at Walmart were challenging and filled with tough learning situations. But as the technology grew, and as Walmart's experience grew with it, what emerged is a true blend of art and technology that provides availability and scalability that is designed to keep the systems running.

1.5.1 Walmart's experience

Similar to most clients, Walmart started using their Parallel Sysplex to gain the ability to scale as they began outgrowing the largest IBM Z server at the time. Although they "walked" into a base Parallel Sysplex architecture, they "ran" head first into IBM DB2® data sharing with their eye on the prize of growth. The CF was used for XCF, GRS, and group buffer pool structures, as was standard in the early days of this technology.

And grow they did! By 2007, Walmart was running 200 million CICS transactions a day, mostly with 3270, Systems Network Architecture (SNA), data sockets, and MQ. In 2009 Walmart introduced CICS Web Services, and by 2011 they had grown to 500 million CICS transactions a day. The most recent changes were in 2013 when Walmart began using Representational State Transfer (REST) services. Walmart's first production REST service, a caching service, was implemented in 2013 and quickly followed by a NoSQL key value database and a unique ID generator. As Walmart began adding modern CICS services, their transactions stabilized in the traditional environments and grew quickly in the new services area. The growth was so fast that the environment was separated into its own sysplex to allow for more isolation, a light weight stack, and a more agile environment.

Figure 1-1 shows a summary of the Walmart CICS transaction growth.

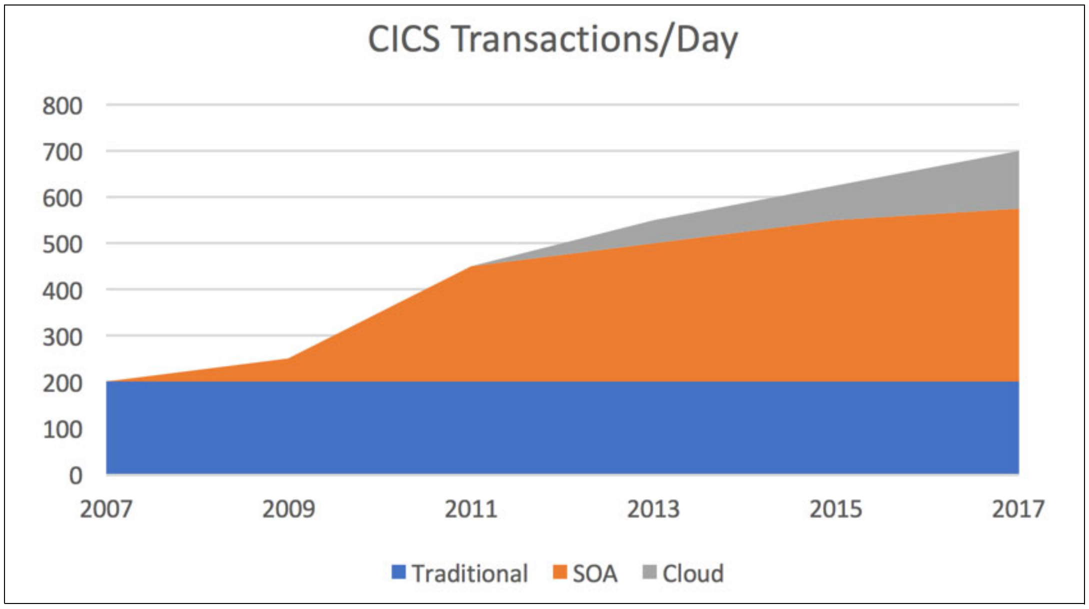


Figure 1-1 Walmart CICS transaction per day growth

1.5.2 Walmart's references

Walmart has been bold in sharing both their experiences and code. They have provided a view for a practical approach to getting started with cloud services in *Creating IBM z/OS Cloud Services*, SG24-8324. You can find further documentation about using CICS to create cloud services from the personas involved in *How Walmart Became a Cloud Services Provider with IBM CICS*, SG24-8347.

Walmart is a leader in contributions to the open source world and has provided the following of their z/OS cloud services in GitHub:

- zUID: z/OS-based unique identifier generator
- zECS: z/OS-based enterprise cache service
- zFAM: z/OS-based file access manager

Using the Coupling Facility lets us use shared resources to compliment the breadth of our application processing without all the complexities of true distributed computing. —
Rich Jackson, Principle Systems Engineer, Walmart

Walmart encourages interaction and comments to provide further innovation for these services. Walmart chose a method to provide cloud services and to fully use the Parallel Sysplex environment to drive benefits for their user community. But this method is not the only way to achieve success. Your experiences and goals might drive you down a different path or a different way to use the Coupling Facility. We encourage you to share your experiences and contribute to the overall innovation of the platform.

1.6 Moving forward

What follows in subsequent chapters is a description of uses of Coupling Facility structures and functions, both as they were meant to be used and as Walmart has found them useful in their environment. Chapters 2 through 6 give an overview of elements that can be used with the CF, some information about setup and APIs, and descriptions of creative ways that Walmart is using the CFs in their environment. Chapter 7, “Combined functional use” on page 57 provides an advanced view of Walmart’s usage with an explanation of what can be achieved by combining several of the elements and applying them to real business problems.

2

VSAM record-level sharing

Record-level sharing (RLS) is a Virtual Storage Access Method (VSAM) function that was first provided by DFSMS z/OS V1.3 and that is used by IBM CICS Transaction Server (CICS TS) for z/OS V1.1 and higher. VSAM data sets are opened in RLS mode, which allows them to be shared (with full update capability) among applications that are running in multiple CICS regions within a sysplex.

In Data Facility Storage Management Subsystem (DFSMS) z/OS V1.3, support was implemented for a new data sharing subsystem, SMSVSAM, which runs in its own address space. SMSVSAM provides the VSAM RLS support that is required by CICS application-owning regions and batch jobs within each z/OS system image in a Parallel Sysplex environment. The SMSVSAM subsystem, which is generally initialized automatically during an z/OS initial program load, uses the IBM Coupling Facility (CF) for its cache and lock structures. It also supports a common set of buffer pools for each z/OS image.

This chapter provides a high-level overview of the VSAM RLS function and its uses. It includes the following topics:

- General use of VSAM RLS in CICS
- Setting up VSAM RLS in CICS
- Uses of VSAM RLS in CICS

2.1 General use of VSAM RLS in CICS

Before RLS, CICS users could share VSAM data sets with integrity by using *function shipping* to a file-owning region. With function shipping, one CICS region accesses the VSAM data set on behalf of other CICS regions. Requests to access the data set are shipped from the region where the transaction is running to the region that has access to the file.

Function shipping provides a solution for the CICS user, but it has limitations. For example, function shipping does not address the problems of sharing data sets between CICS regions and batch jobs.

Note that the file owning region (FOR) is constrained to a single task control block (TCB) unless using an IPIC connection, and therefore is limited to the speed of a single central processor (CP). RLS is not subject to this constraint because all of the work is done across multiple application owning regions (AORs) where the application resides.

VSAM RLS provides benefits in terms of high availability, lower overhead, and scalability over non-RLS VSAM file access that uses CICS FORs. Conversion to RLS also provides the opportunity to remove FORs and, thus, a frequently cited single point of failure.

2.2 Setting up VSAM RLS in CICS

The enablement of VSAM RLS at the sysplex and z/OS system level is outside the scope of this book; however, this section provides the following summary information:

- Changes to CICS parameters to support RLS
- Altering file definitions to enable RLS access
- Application considerations when using RLS

2.2.1 Changes to CICS parameters to support RLS

To enable an individual CICS region to use VSAM RLS, specify the following system initialization table (SIT) parameter:

```
RLS=YES
```

Also review the current settings of the following SIT parameters:

- `NONRLSRECOV=`
- `FCQRONLY=`

2.2.2 Altering file definitions to enable RLS access

So that files can be accessed in RLS mode, make the following changes to data set and file definitions:

- Confirm that the data set type and attributes make it eligible for RLS access.
- Define or alter the integrated catalog facility (ICF) catalog file entry to specify a valid `LOG` attribute.
- Change the `FILE` resource definition to specify `RLSACCESS(YES)`.
- Change the `FILE` resource definition to specify a suitable value for the `READINTEG` attribute. The default `UNCOMMITTED` corresponds to non-RLS local shared resource (LSR) access.

Access to a VSAM data set: Collective access to a VSAM data set must be RLS or non-RLS. CICS regions cannot simultaneously open a file in RLS and non-RLS mode.

2.2.3 Application considerations when using RLS

Although the CICS API commands that are required to access VSAM files are the same for RLS and non-RLS access, application designers need to be aware that the response codes, locking behavior, and recovery scenarios are different when accessing the file in RLS mode. Review the CICS documentation in IBM Knowledge Center and perform thorough testing as part of the conversion process.

2.3 Uses of VSAM RLS in CICS

VSAM RLS is a CF technology that doesn't require a lot of custom code to use after it has been set up. So, although it is a useful CF-based technology that warrants attention, this section does not include specific examples of its use. The focus for RLS occurs more around higher-level design considerations. This section explores some of these types of considerations.

2.3.1 Eliminating file owning regions

The ability to directly share the VSAM data source among AORs provides some significant benefits. It simplifies the topology and improves availability by removing the dedicated region for shared data interactivity.

Although the full setup for VSAM RLS is beyond the scope of this publication, several items are worth mentioning to ensure that they are considered for an optimal implementation:

- The `RLS_MaxCFFeatureLevel` system parameter and the RLS CF `CACHE` data class parameter

  These settings determine which particular VSAM structure components are cached during processing.
- Sizing for buffer pools, lock structures, and cache structures

  These settings can greatly affect RLS performance and should be considered carefully. The appropriate sizing calculation for these items are generally dependent on other system settings.
- System managed assignments

  You can use DFSMS to associate VSAM RLS files with particular cache structures.

2.3.2 Improved scalability

Complementary to the simplicity and availability improvements that are provided by replacing the FOR with VSAM RLS is the additional scalability that it enables. With the removal of the single-threaded FOR operations, the processing of data access can now be distributed across numerous CPUs on multiple systems.

Many of the examples in this publication are used in conjunction with RLS particularly for this characteristic. In fact, all of the examples herein that use data set storage use RLS. However, additional details are relevant to this design and should be considered.

The main usage pattern for these services is primarily (though, not exclusively) expected to be for web or HTTP workloads. To take full advantage of the benefits that RLS provides, ensure that workloads are distributed to the groups of data sharing regions as the requests come into the system. Instead of the traditional approach of using web owning regions (WORs) to accept traffic and pass it on the AORs, this design combines the two roles into each region in a related cluster. We refer to these regions as *service owning regions* (SORs).

Another CF-related feature, although it's not specifically given focus in this book, is used to distribute requests to these SORs. The Workload Manager (WLM) managed sysplex distribution with dynamic virtual IP address (DVIPA) is used to intelligently distribute workload across the sysplex to the SORs. This process allows you to fully use the processing resources and take advantage of the shared data access that RLS provides. See Figure 2-1.

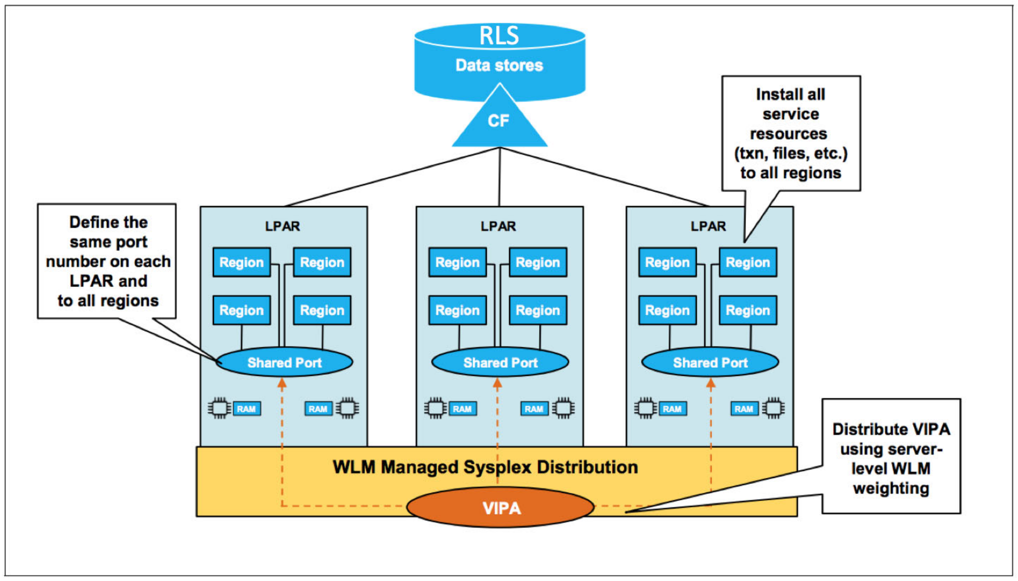

RLS
Data stores
CF
Install all service resources (txn, files, etc.) to all regions
Define the same port number on each LPAR and to all regions
LPAR
Region
Shared Port
RAM
WLM Managed Sysplex Distribution
VIPA
Distribute VIPA using server-level WLM weighting

Figure 2-1 WLM Managed Sysplex Distribution to use RLS

Examples related to online object stores are referenced throughout this publication, and these examples rely upon this system design. The scalability of this design has been proven in real-world production workloads.

This design enabled Walmart to absorb workloads during peak events of six to eight times daily average processing without the need to make configuration changes. Even during average processing days, about one billion I/Os are processed by VSAM RLS. You can find additional details about the RLS system setup, SORs, and WLM managed sysplex distribution that are described in this section in *How Walmart Became a Cloud Services Provider with IBM CICS*, SG24-8347.

3

Named counters

CICS provides a Parallel Sysplex wide facility for the generation of unique sequence numbers, which can be shared by tasks that are running anywhere within a sysplex. Each number sequence, referred to as a *counter*, is accessed by a common name, called a *named counter*. Multiple counters can be defined and logically grouped into pools with each pool of counters mapping to a coupling facility (CF) list structure. Pools provide a logical means of grouping and separating counters. This is useful for individual CICS applications or groups of CICS regions which want dedicated access to counters.

For each pool, counter access is managed by a dedicated server address space. All CICS regions that need to access a pool of named counters do so by communicating with the *named counter server*. Only the named counter server accesses the CF list structure and is responsible for managing the counters within the pool.

Because named counters are stored in the CF, they persist for as long as the CF list structure is allocated. Failure of a task, CICS region, or named counter server does not result in counters being lost. However, in the event of a CF structure failure, the counters are lost. The unlikely failure of a CF and subsequent loss of a list structure can be minimized using system managed duplexing; however, applications should consider the potential loss of named counters in their design.

CICS provides an API and callable interface to create, access, and delete named counters. The application is responsible for defining the named counter along with the minimum and maximum range and initial value of the counter.

This chapter provides an overview of named counters and how to use them effectively. It includes the following topics:

- Common uses of named counters
- Setting up named counters
- Examples of named counter use

3.1 Common uses of named counters

Named counters are frequently used by applications that run in a sysplex environment because they are efficient and universally accessible. Some common examples of how named counter unique sequence numbers are used are:

- Forming part of an account, order, trade, or log sequence number
- Forming part of a more randomized VSAM key-sequenced data set (KSDS) key
- As a basis for routing decisions

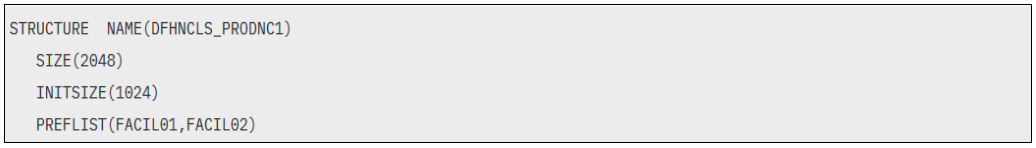
3.2 Setting up named counters

This section summarizes the steps that are required to set up a named counter server and to update CICS to access the server. A sample set of API commands to create and access a named counter server are also provided. You can find full details about setting up and running a named counter, refer to IBM Knowledge Center.

3.2.1 Defining resources to the CF

Each named counter pool requires a CF list structure to be defined. The definition is created by updating the CF resource management (CFRM) policy using the IXCMIAPU utility. The name of the structure is `DFHNCLS_<poolname>`. In addition to the name of the structure, space information is required. IBM provides a tool to assist with the space estimation.

Figure 3-1 shows a sample CFRM policy structure definition that supports a named counter pool called `PRODNC1`.



```
STRUCTURE  NAME(DFHNCLS_PRODNC1)
   SIZE(2048)
   INITSIZE(1024)
   PREFLIST(FACIL01,FACIL02)
```

Figure 3-1   Example of a CICS named counter structure definition

3.2.2 Creating the named counter server

A CICS region can simultaneously access named counters in multiple pools. However, for each pool, a dedicated named counter server is required on each LPAR where access is required. You can find detailed instructions for setting up the server setting up the server, including information about configuration and security, in IBM Knowledge Center.

Example 3-1 shows an example of the named counter server JCL for the `PRODNC1` pool.

Example 3-1   Named counter server sample JCL

```
//CICSNC1 JOB …
//NCSERVER EXEC PGM=DFHNCMN,REGION=32M,TIME=NOLIMIT
//STEPLIB   DD  DISP=SHR,DSN=CICS.SDFHAUTH
//          DD  DISP=SHR,DSN=CICS.SDFHLIC
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD  *
POOLNAME=PRODNC1
/*
```

3.2.3 Changes to CICS to access the named counters server

No changes to CICS are required to support the named counter server access as all **`COUNTER`** and **`DCOUNTER`** API commands allow the `poolname` parameter to be specified using the `POOL` option. However, CICS does provide a named counter options table, called DFHNCOPT, which you can create and which allows you to map a logical pool name to the actual pool name. This mapping is a particularly useful mechanism for allowing test and production regions to use a different counter pool without needing to change the pool name that is specified by the application programs.

You can find details about defining and using the named counter options table defining and using the named counter options table in IBM Knowledge Center.

In the event no pool name is specified on the API command line, CICS uses the `NCPLDFT` system initialization parameter value as the default pool name.

3.2.4 Creating and accessing a named counter

CICS provides access to named counters using the following methods:

- CICS API commands
- Callable interface

Note: The callable interface allows non-CICS applications to use named counters.

Tasks running in CICS typically use the API interface to define and access named counters. Table 3-1 provides an overview of the commands.

Table 3-1 API commands for accessing named counters

API command	Description
`DEFINE COUNTER`	Define a counter to the pool.
`GET COUNTER`	Get the next sequence number from the named counter.
`QUERY COUNTER`	Get the current, minimum, and maximum value of the named counter.
`UPDATE COUNTER`	Change the current sequence number of the counter to a specified value.
`REWIND COUNTER`	Reset a counter that has reached its maximum limit to its minimum limit.
`DELETE COUNTER`	Delete the named counter.

The **`COUNTER`** command can be replaced with the **`DCOUNTER`** command if 64-bit named counters are required.

Figure 3-2 illustrates a sample scenario where two CICS tasks are running in different CICS regions, thus creating and accessing a named counter. Each **GET COUNTER** request returns a unique counter value.

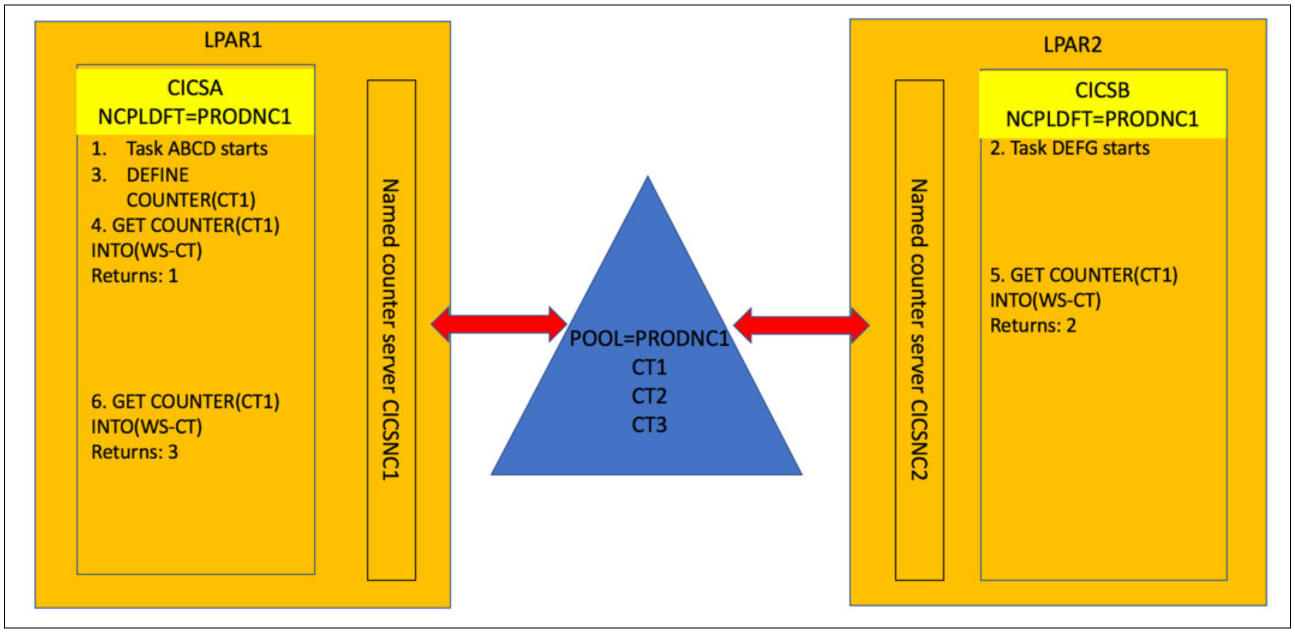

LPAR1
CICSA
NCPLDFT=PRODNC1
1. Task ABCD starts
3. DEFINE COUNTER(CT1)
4. GET COUNTER(CT1) INTO(WS-CT) Returns: 1
6. GET COUNTER(CT1) INTO(WS-CT) Returns: 3
Named counter server CICSNC1
POOL=PRODNC1
CT1
CT2
CT3
LPAR2
CICSB
NCPLDFT=PRODNC1
2. Task DEFG starts
5. GET COUNTER(CT1) INTO(WS-CT) Returns: 2
Named counter server CICSNC2

Figure 3-2 Flow of multiple tasks accessing a named counter

The minimum, maximum, and initial value of the counter can be defined on the **DEFINE COUNTER** call. The counter can also be made to wrap when the maximum value is reached. If a task requires multiple sequence numbers, use the `INCREMENT` keyword on the **GET COUNTER** request.

3.3 Examples of named counter use

The following sections illustrate scenarios where named counters are used to address real-world needs.

3.3.1 Generating a unique ID

For this scenario, a requirement surfaced for the local generation of a Universally Unique Identifier (UUID) on a z/OS system, as newer versions of the operating system no longer provide support for this function. It was determined that strict adherence to RFC 1422, which describes UUID, was not necessary. Therefore, a method was developed to generate unique IDs with the same format as the common UUID.

A new algorithm was developed, and a named counter was used to generate a portion of the UUID. The components of the UUID comprise of a *site ID*, which is static, a *time value* that perpetually moves forward, and a *sequential number*, which increments based on demand. Each component is represented in hex and concatenated together, as illustrated in Figure 3-3 on page 15.

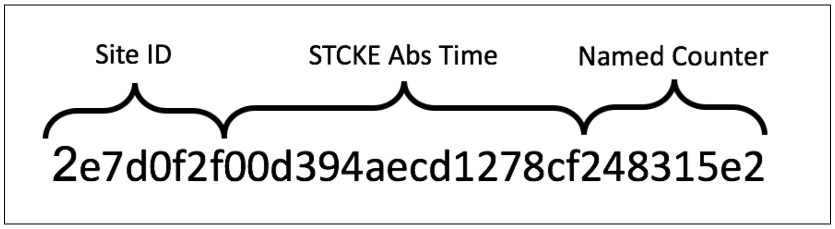


Figure 3-3 Concatenation of UUID components

After the value is generated, it is formatted with dashes to resemble and match the UUID format, as illustrated in Figure 3-4.

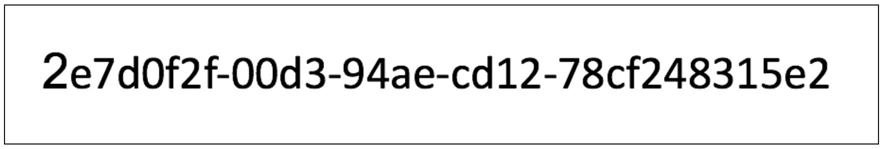

2e7d0f2f-00d3-94ae-cd12-78cf248315e2

Figure 3-4 Formatted to resemble a standard UUID format

Access to the new UUID generator is provided through a web service call, making it available to any client in the network, not just local z/OS processes. The code for this service is available for review or use from GitHub.

3.3.2 Generating components of keys

The sequential number generation provided by named counters can be used by parallel processes to generate unique values. Another use of this feature involves the generation of keys to identify related data assets.

This example relates to an online object store that supports large objects of up to 2 GB in size. The object storage capability is accomplished by segmenting the object and storing the pieces across a collection of VSAM KSDSs. In a simple example, the *key* (or name) of an object is stored in one KSDS, and the object data is segmented into 32 KB chunks and stored in one or more additional KSDSs.

The association between the object key and the data segment or segments is maintained using a type of *pointer* (internal key) that is imbedded in each data segment. Named counter services are used in the generation of the unique pointer for the data store.

The objects that are stored can contain structured or unstructured data. For structured data, the service provides the capability to define the data structure and establish additional indexes associated with certain fields within the object. As the indexes are defined, they are stored in the additional KSDS files, and additional pointers are created to maintain the relationship between the various objects.

The internal keys comprise of several concatenated components, not all of which are necessarily relevant to this publication. Within the context of the named counter, one component is relevant. The first component of the internal key is an absolute time value acquired using the z/OS `STCKE` instruction. To ensure the uniqueness of this portion of the internal key, a half-word binary named counter value is appended to the time component. Other components, consisting of metadata related to the object segments, are then appended to the time and the named counter values to complete construction of the internal key.

The internal keys are then used to associate the keys, indexes, and various segments of each object entry, as illustrated in Figure 3-5.

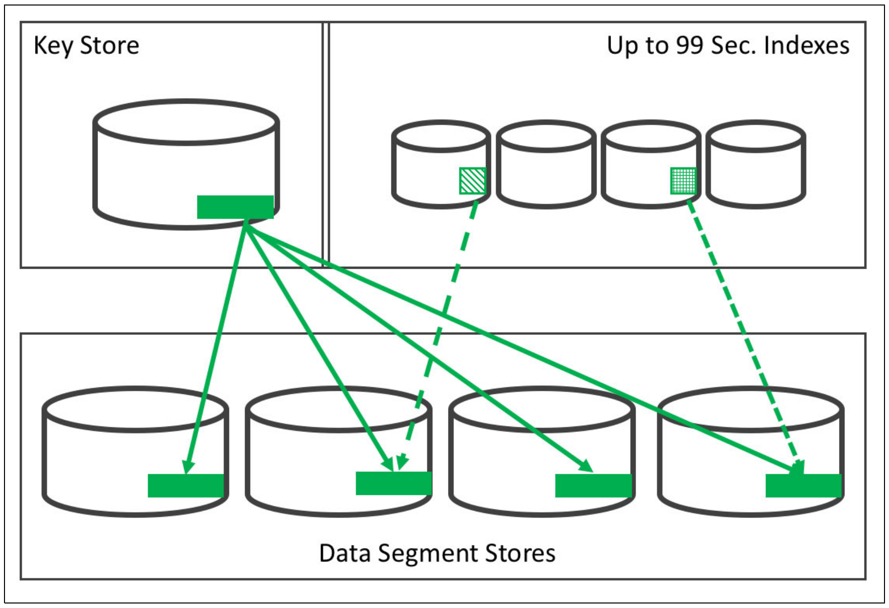


Figure 3-5 Internal keys associated with various object segments

The source code that is associated with this product is available for review or use on GitHub. The following portions are relevant:

- The HLASM code to define a named counter instance
- COBOL code to get a new value from the named counter

3.3.3 Distributing the workload and resources

An auto-incrementing counter that wraps or rolls over can be useful in a variety of scenarios. In this example, a named counter is accessed by parallel processes to help control and balance the distribution of workload across a group of clustered resources.

A product was developed to capture and aggregate log messages that were being written to a local transient data queue (TDQ) across numerous disparate CICS regions. The solution consists of a CICS global user exit that was deployed into each CICS regions (*source regions*). The global user exit captures the log messages and passes them over a set of Internet Protocol interconnectivity (IPIC) connections to TDQs in another set of message processing CICS regions (*target regions*). The target regions include tasks that asynchronously process the messages and issue web client requests to pass them to a cloud service endpoint.

To balance the workload across the target regions, the global user exit was designed to dynamically build its `WRITEQ TD` command based on the following components:

- A predefined 2-character prefix for the SYSID that is associated with the group of target regions
- A 2-digit numeric value that is acquired from a wrapping named counter with a maximum value that matches the number of target regions

The SYSID for each of the target regions (up to 100) are defined following this convention, as listed in Table 3-2.

Table 3-2 Naming convention

Target region	**SYSID** **{SYSID Prefix}{2-digit NC}**
CICS00	SY00
CICS01	SY01
CICSnn	SYnn
CICS99	SY99

The SYSID prefix, which is stored in the global user exit global work area (GWA) is predefined. In combination with the named counter value, the dynamically constructed SYSID is added to the **`WRITEQ TD`** command at execution time.

This process provides flexibility in deploying and managing the global user exit. The TDQ is defined with the same name in all target CICS regions, allowing the name component of the **`WRITEQ`** command to remain static. So, the only thing that changes on each **`WRITEQ`** command is the SYSID.

With this design, each invocation of the **`WRITEQ TD`** command routes the message to the next target region in the cluster, thereby providing a round-robin effect for each invocation of the command, as illustrated in Figure 3-6.

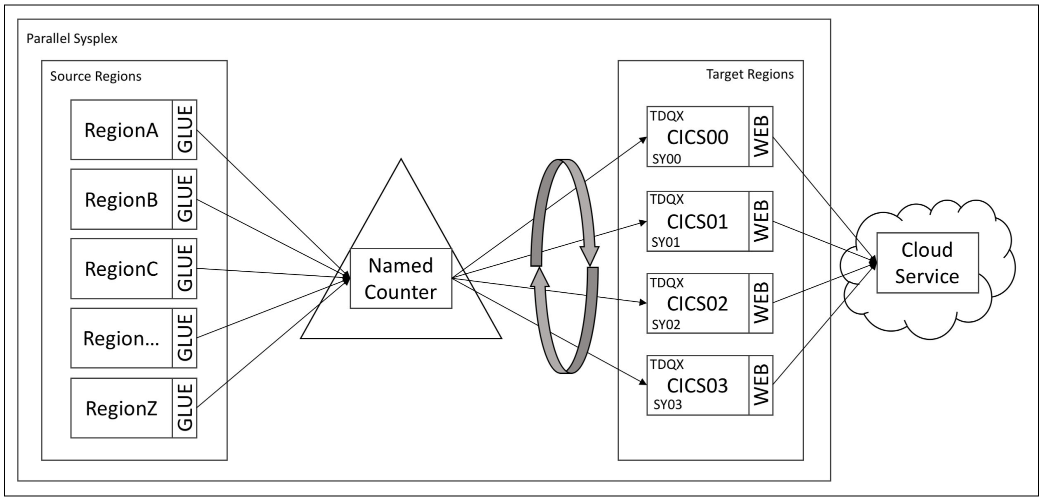


Figure 3-6 Distribution of workload

Because all source regions across the Parallel Sysplex acquire their 2-digit numeric suffix for the target SYSID from a single shared resource (the named counter), all writes are equally distributed across the target regions regardless of the quantity of source regions, source TDQ message volume, or overall workload.

3.3.4 Tracking activity

Sometimes, a good use of a named counter is simply to count. There is a plethora of possible activities that can be of benefit to track. If these activities occur within CICS, especially groups of sysplex-distributed CICS regions, a named counter provides an excellent mechanism for quantifying the volume of these activities.

One example involves web traffic. Although System Management Facilities (SMF) capture a rich collection of details that are related to TCP/IP activity on the system, some relevant details, particularly at or above the presentation layer of network communications, are not included in this data capture. Just as read/write ratios are important for analyzing I/O operations, the distribution of HTTP methods in web traffic can be useful for managing online workloads. However, this information is not currently captured in traditional sources such as SMF.

The primary HTTP methods are directly relatable to basic create, retrieve, update, and delete operations, as listed in Table 3-3.

Table 3-3 HTTP methods for the create, retrieve, update, and delete operations

HTTP method	**Operation**
POST	Create
GET	Retrieve
PUT	Update
DELETE	Delete

If considering this method from the same perspective as a database administrator considers basic create, retrieve, update, and delete operations, it’s not difficult to fathom why tracking HTTP method volumes might be useful. A workload that is heavily POST (that is write/create) oriented might need to be configured and managed differently than a workload that is heavily GET (that is, read) oriented.

Although this type of information cannot be captured by SMF, processes running in CICS can relatively easily log this activity with named counters. If other HTTP methods, such as HEAD or PATCH, are used in meaningful amounts, they can be tracked as well. You just need to set up the appropriate counters in association with the resource to be tracked and then tally based on activity.

This example defines a named counter for each HTTP method of each new HTTP service instance deployed. For consistency, each counter is named by appending the HTTP method to the transaction ID that is associated with the HTTP service, as listed in Table 3-4.

Table 3-4 Counter names

Counter Names {Trans ID}{HTTP method}
TXN1POST
TXN1GET
TXN1PUT
TXN1DELETE
TXN1HEAD

For convenience, the CONVERTER program in the URIMAP for an HTTP service can be used to tally the activities that are associated with that service. It is used to extract the method from the incoming request and to increment the associated counter.

Example 3-2 and Example 3-3 demonstrate the setup and execution of incrementing the counters.

Example 3-2   HTTP METHOD counter resources

```
*****************************************************************
* HTTP METHOD Counter  resources.                               *
*****************************************************************
01  HTTP-METHOD-LENGTH      PIC S9(08) COMP VALUE 10.

01  HTTP-METHOD             PIC  X(10) VALUE SPACES.
01  HTTP-POST               PIC  X(10) VALUE 'POST      '.
01  HTTP-GET                PIC  X(10) VALUE 'GET       '.
01  HTTP-PUT                PIC  X(10) VALUE 'PUT       '.
01  HTTP-DELETE             PIC  X(10) VALUE 'DELETE    '.
01  HTTP-HEAD               PIC  X(10) VALUE 'HEAD      '.

01  NC-HTTP-METHOD.
   02  NC-TRANID           PIC  X(04) VALUE SPACES.
   02  FILLER              PIC  X(01) VALUE '_'.
   02  NC-METHOD           PIC  X(10) VALUE SPACES.
   02  FILLER              PIC  X(01) VALUE SPACES.

01  NC-VALUE                PIC  9(16) COMP VALUE 0.
01  NC-INCREMENT            PIC  9(16) COMP VALUE 1.
```

Example 3-3   Extract HTTP method and HTTP method counter

```
*****************************************************************
* Extract HTTP Method.                                          *
*****************************************************************
0200-EXTRACT-METHOD.
   EXEC CICS WEB EXTRACT
      HTTPMETHOD  (HTTP-METHOD)
      METHODLENGTH(HTTP-METHOD-LENGTH)
      NOHANDLE
   END-EXEC.

0200-EXIT.
   EXIT.

*****************************************************************
* Increment HTTP Method counter.                                *
*****************************************************************
0300-INCREMENT-COUNTER.
   MOVE HTTP-METHOD    TO NC-METHOD.
   MOVE EIBTRNID       TO NC-TRANID.

   EXEC CICS GET
      DCOUNTER (NC-HTTP-METHOD)
      VALUE    (NC-VALUE)
      INCREMENT(NC-INCREMENT)
      NOHANDLE
   END-EXEC.

0300-EXIT.
   EXIT.
```

With these components in place, it is now possible to track the characteristics of an HTTP workload based on the distribution of HTTP method types, as illustrated in Figure 3-7.

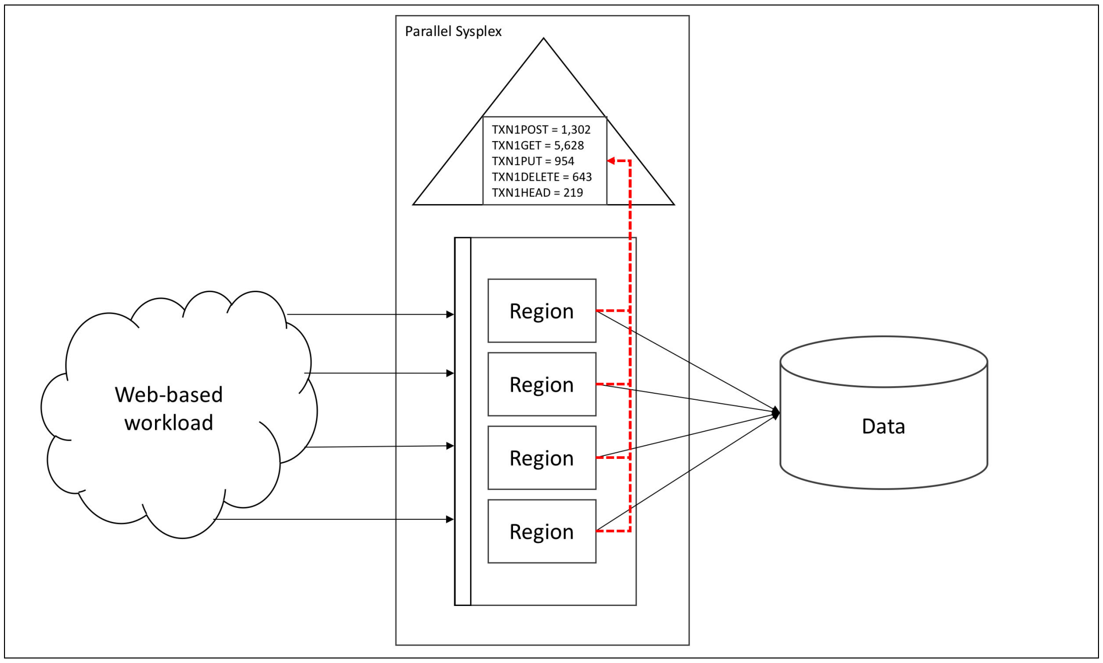


Figure 3-7 Tracking HTTP method ratios

Although this example describes tracking HTTP methods that are used at an individual service or transaction level, you can apply the same approach to track the information at the region level or even at the sysplex level. You can make slight modifications to the implementation described here to make that possible.

It is worth noting that this approach is related to purer HTTP activity, such as RESTful services. One of the many benefits of these types of services is the ability to distinguish the intent or purpose of a transaction from the message format, instead of from application fields typically used in the SOAP request/response model or 3270 send/receive fields. Having this visibility and using it as described provides a level of insight not currently available with this type of workload.

4

Global resource serialization

In a multitasking, multiprocessing environment, *resource serialization* is a technique that is used to coordinate access to resources that are used by more than one program. When multiple users share data, a way to control access to that data is needed. Users who update data, for example, need exclusive access to that data; if several users tried to update the same data at the same time, data integrity exposure (the possibility of incorrect or damaged data) would result. In contrast, users who only read data can safely access the same data at the same time.

The z/OS system provides multiple ways of providing serialized access to data on single or multiple systems, but global resource serialization (GRS) is a fundamental way for programs to get the control they need and ensure the integrity of resources in a multisystem environment.

When considering the serialization of resources, keep in mind the following factors:

- The resource name: There must be a consistent name for the resource being serialized. Convention dictates that resource names are made up of a major and minor name. The former generally describes the type of resource, and the later names the resource.
- The scope of the serialization: Scope determines how visible the serialization is. The scope can be limited to a task, address space, or system wide.
- The duration of serialization: After serialization, resources are normally held until they are explicitly released. In some instances, it might be advantageous for resources to be automatically released when a certain event occurs.

In a CICS context, applications do not normally need to manage the serialization of resources they access. Serialization of files, databases, and queues is performed by CICS and the underlying features they access. However, sometimes tasks need a mechanism to serialize access to user-defined resources. CICS provided the **ENQ** and **DEQ** API commands to manage resource serialization. By default, the scope of a CICS ENQ is the unit of work (UOW). Thus, when CICS performs a sync point, the serialization is released automatically. The scope of serialization is normally limited to the CICS address space, although using GRS, CICS can perform sysplex-wide resource serialization.

This chapter outlines how GRS is achieved in CICS and shows examples of its use by tasks that are running in a sysplex environment.

4.1 Common uses of CICS global enqueues

A CICS global enqueue is frequently used by CICS applications that execute in multiple CICS regions to complete the following types of tasks:

- Serialize the updating of a common resource
- Synchronize an activity over multiple regions
- Inhibit a process or procedure from executing
- Notify other instance of an application that a process is in progress

4.2 Setting up global enqueues in CICS

Consider the scenario in which a generic CICS application that is currently running in a single CICS system is to be altered so that it can run in multiple CICS regions across a sysplex. By design, the application currently uses an in-memory table to store frequently accessed dynamic data. The table resides in the common work area (CWA) that is easily addressed by all programs in the application.

It might be necessary to periodically update the table. the technique used to avoid concurrent updates is achieved using the CICS **`ENQ`** and **`DEQ`** commands. The task that decides to update the table performs the following sequence:

1. Issue the **`EXEC CICS ENQ RESOURCE(WS-RESOURCE) LENGTH(10)`** command.
2. Construct the new in-memory table contents.
3. Write data to the CWA.
4. Issue the **`EXEC CICS DEQ RESOURCE(WS-RESOURCE) LENGTH(10)`** command.

Note: The `WS-RESOURCE` keyword contains a value of `'CICSAAS-IT'`.

If all tasks updating the in-memory table follow the same convention and specify the same resource name on the **`ENQ`** and **`DEQ`** commands only one task will be able to update the table at any one time.

Note: The **`ENQ`** and **`DEQ`** commands have a scope of the CICS system in which it is issued and the **`ENQ`** command duration is by default UOW. Thus, if the task fails to issue the **`DEQ`** command, it is releases automatically at the next implicit or explicit sync point. (An implicit sync point occurs at task termination.)

The application designer now decides to move the in-memory table into a CICS Coupling Facility Data Table (CFDT).

To avoid the multiple update issue, the current **`ENQ`** and **`DEQ`** mechanism is retained, but it is necessary to change the scope to be a system-wide **`ENQ`** command. CICS supports system-wide **`ENQ`** and **`DEQ`** commands using the `ENQMODEL` resource. A new resource definition is created using the following command:

```
DEFINE ENQMODEL(APP1ENQ) GR(APP1GRP) ENQNAME(CICSAAS-IT) ENQSCOPE(REDB)
```

The specification of a non-blank `ENQSCOPE` keyword alters the enqueue from being a CICS-managed enqueue with a scope of the local CICS system to a GRS-managed **`ENQ`** with sysplex-wide scope.

A benefit of using the `ENQMODEL` keyword to change the scope of the **`ENQ`** command is that no changes to existing applications are necessary. The application then changes the in-memory table creating process to update a CFDT entry as follows:

1. Issue the **`EXEC CICS ENQ RESOURCE(WS-RESOURCE) LENGTH(10)`** command.
2. Construct the new in-memory table contents.
3. Write data to the CFDT.
4. Issue the **`EXEC CICS DEQ RESOURCE(WS-RESOURCE) LENGTH(10)`** command.

Note: The `WS-RESOURCE` keyword contains a value of `'CICSAAS-IT'`.

You can find a full overview of sysplex enqueue and enqueue in IBM Knowledge Center.

4.3 Use cases for CICS global enqueues

It's important to note that an enqueue can be placed on virtually any named resource that exists within the operating environment. This capability opens the door to a wide variety of scenarios where global enqueues can be used for gain. This section delves into some applicable examples of using global enqueues to improve or expand on current capabilities.

4.3.1 Global transaction class

The CICS `TRANCLASS` resource is useful in controlling how a workload is managed. The ability to control the number of currently dispatched tasks provides an administrator with controls to maintain stability and desired workflow within their environment. However, the `TRANCLASS` resource is relevant only in the context of each single region. If distributing work across numerous regions in a sysplex, it can be useful to control this at the sysplex level instead of by relying upon a simple aggregation of the totals of the individual `TCLASS` attributes for all the regions.

Capabilities similar to the region-level `TCLASS` can be achieved at the sysplex-level by using GRS. To enable this concept of a global trans class, the resource must first be moved to the global level. This is accomplished, as outlined in 4.2, "Setting up global enqueues in CICS" on page 22, by defining an `ENQMODEL` with an appropriate ENQSCOPE specified.

A predetermined maximum value of global tasks needs to be established, and then the **`EXEC CICS ENQ RESOURCE()`** command can be issued by a dispatcher program on the resource name. A "dispatcher program" in this context refers to a task that is invoked (or LINKed to) from an initiating program, such as an upstream transaction program or a CONVERTER that is associated with an HTTP request, and handles the delegation of related downstream tasks. The resource name will be some value (any value, really) with a numeric suffix. The program then issues the ENQ against the resource name while incrementing the numeric suffix. When an ENQ is successful, the task can be started.

If the program reaches the pre-determined maximum value of global tasks before acquiring a successful ENQ, the task is not submitted. In this case, the program will then, ideally, need to restart the process of checking for ENQ. This is a suitable, and eloquent, approach when utilizing background tasks and interval control element (ICE). Otherwise, some type of response might need to be returned to the client if the tasks are 3270 or HTTP based. The response should indicate that the maximum number of tasks has been reached. An HTTP status code 429 would be applicable in this scenario, or a custom response for 3270 tasks. See Figure 4-1 on page 24.

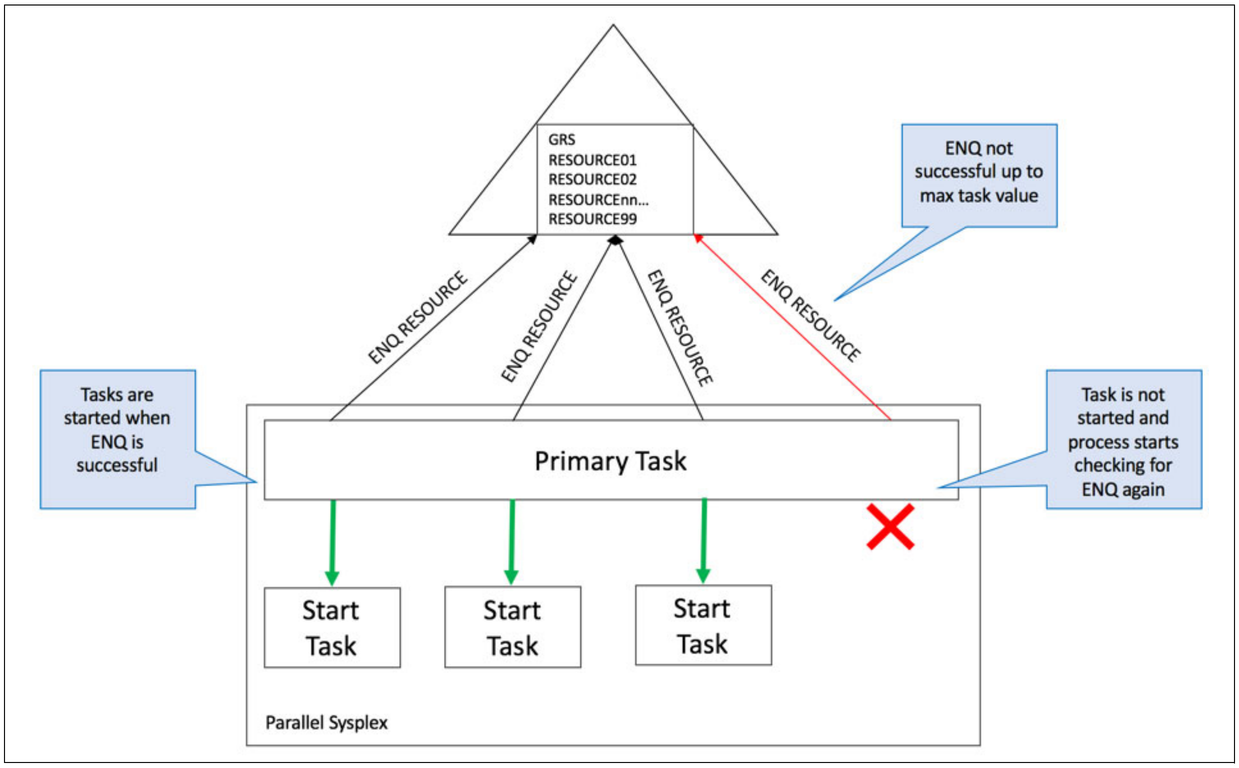


Figure 4-1 GRS ENQ determines task start

This approach allows us to control the total number of active tasks across the entire sysplex. Example 4-1 samples demonstrate the basics for the dispatcher program to define and check for the enqueue.

Example 4-1 ENQ-DISPATCHER

```
01  MAXACTIVE-DISPATCHER    PIC  9(02) VALUE 20.

*****************************************************************
** ENQ Dispatcher Level                                        **
*****************************************************************
 01   ENQ-DISPATCHER.
     02  FILLER              PIC  X(08) VALUE 'CICSGRS_'.
     02  FILLER              PIC  X(04) VALUE 'OMS_'.
     02  FILLER              PIC  X(11) VALUE 'DISPATCHER_'
     02  DISPATCHER-COUNT    PIC  9(02) VALUE ZERO.


2500-ENQ-DISPATCHER.

     PERFORM WITH TEST AFTER
         VARYING DISPATCHER-COUNT   FROM 1 BY 1
         UNTIL   DISPATCHER-COUNT  EQUAL MAXACTIVE-DISPATCHER
         OR      EIBRESP EQUAL DFHRESP(NORMAL)

         MOVE ENQ-OM91-COUNT TO ENQ-OM91-Suffix

         EXEC CICS ENQ RESOURCE(ENQ-DISPATCHER)
              LENGTH(LENGTH OF  ENQ-DISPATCHER)
              NOSUSPEND
              TASK
```

```
           END-EXEC

       END-PERFORM.

       IF  EIBRESP EQUAL DFHRESP(ENQBUSY)
           PERFORM 8000-RESTART-DISPATCHER   THRU 8000-EXIT
           PERFORM 9000-RETURN               THRU 9000-EXIT.

 2500-EXIT.
       EXIT.
```

4.3.2 Serialization for dispatched tasks

The previous section described a dispatcher task (parent) starting other tasks (child) while using GRS to control the active number of these dispatched tasks across the sysplex. It's a pretty safe assumption that the child tasks need to perform some kind of work. (Otherwise, why start them?) This work might require the task to take ownership of some other resource to ensure there are no conflicts.

We'll use a scenario where the child task needs to process a record (like a purchase order, for example) against a database. To ensure serialization through the full process, the following forms of ENQ are used:

- An ENQ to let dispatcher tasks know that a child task for a particular record is assigned.
- An ENQ to ensure the child task has exclusive access to the actual processing of the record.

The task assignment enqueue is at a unit of work (UOW) scope and can possibly be released during a sync point. This is due to the fact that the dispatcher task can issue a SYNCPOINT to release all enqueues at the same time, instead of relying on a maintained list of enqueue.

The processing enqueue is at a task scope and ensures exclusivity until the processing of the record is completed. During the processing of a record, an occasional sync point is issued (each time some number of updates have been applied as part of the record). The task scope allows the processing task to retain the enqueue, while these sync points are issued, and prevents the dispatcher task from starting another task for processing this particular record. See Figure 4-2 on page 26.

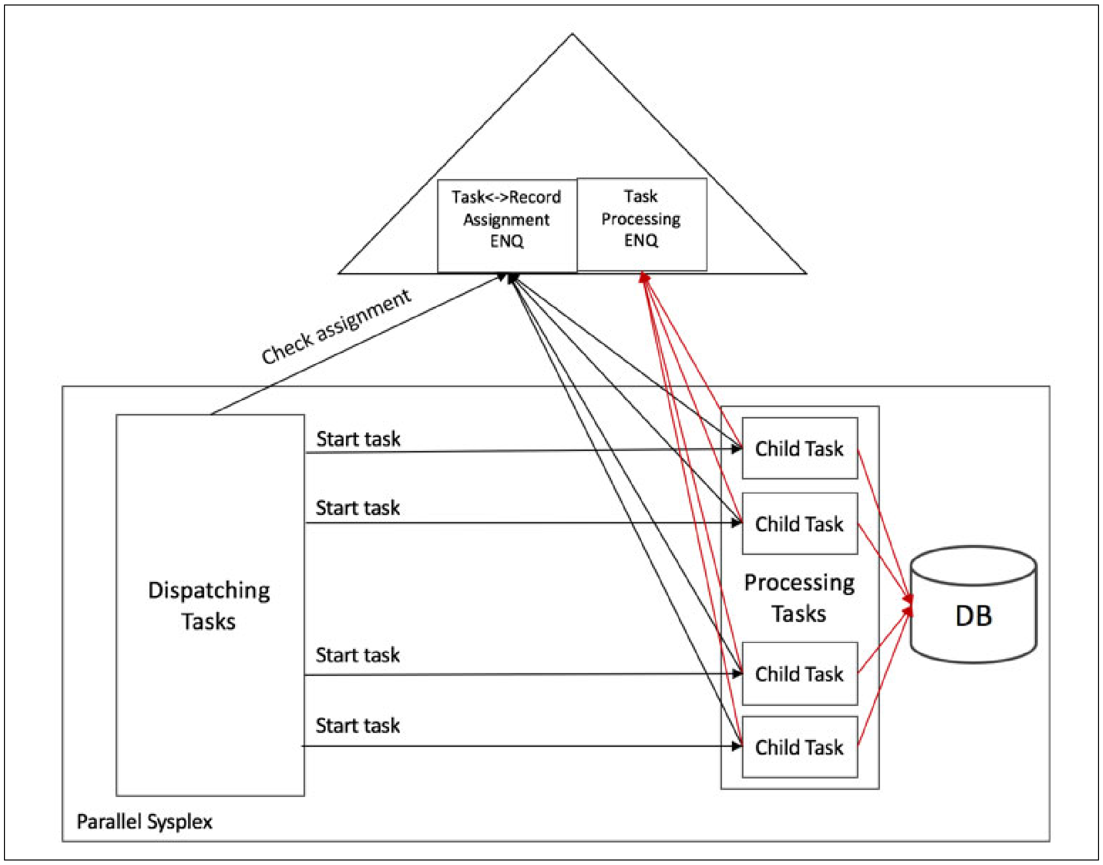


Figure 4-2   ENQs for child tasks

The following code samples demonstrate the steps for establishing the appropriate enqueues for this scenario. Example 4-2 shows the child task perspective.

Example 4-2   Child task perspective

```
*****************************************************************
** ENQ Child CRUD level                                        **
*****************************************************************
 01  ENQ-CRUD.
     02  FILLER              PIC  X(08) VALUE  CICSGRS_'.
     02  FILLER              PIC  X(04) VALUE 'OMS_'.
     02  FILLER              PIC  X(04) VALUE 'CRUD'.

*****************************************************************
** ENQ Child CRUD active level                                 **
*****************************************************************
 01  ENQ-CRUD-ACTIVE.
     02  FILLER              PIC  X(08) VALUE  CICSGRS_'.
     02  FILLER              PIC  X(04) VALUE 'OMS_'.
     02  FILLER              PIC  X(05) VALUE 'CRUD_'.
     02  FILLER              PIC  X(06) VALUE 'ACTIVE'.

*****************************************************************
** ENQ at Child CRUD level.                                    **
** ENQ at Child CRUD active level.                             **
**                                                             **
** The Dispatcher issues an enqueue before attaching the child **
** CRUD task.                                                  **
**                                                              **
** When the Dispatcher gets the ENQ, the child CRUD subtask    **
** is attached, however, there is a gap between DEQ in the     **
** Dispatcher and ENQ in the child CRUD subtask.               **
**                                                              **
** This process will issue both child CRUD and child CRUD      **
** active level ENQ.  When the child CRUD level ENQ has been   **
** acquired, dispatchers won't be able to obtain the lock,     **
** which will prevent other child CRUD subtasks from being     **
** attached.  When the child CRUD active level ENQ has been    **
** acquired, other child CRUD subtasks won't be able to obtain **
** the lock, which will cause the other child CRUD task to     **
** issue a RETURN.                                              **
**                                                              **
** The ENQ is held for the life of the task by explictly       **
** specifying the TASK parameter.  This is required when       **
** processing the CRUD requests, as a SYNCPOINT must be issued **
** at certain frequencies, which would release an ENQ issued   **
** with the UOW scope.                                          **
*****************************************************************

*****************************************************************
** Obtain lock on CRUD active level.                            **
** Read PO from CFDT for each sequence number.                  **
** Apply CRUD requests to database.                             **
** Delete PO records from CFDT for each sequence number.        **
** SYNCPOINT (commit) CRUD request to database.                 **
*****************************************************************
 1000-PROCESS-CFDT-PO.

     EXEC CICS ENQ RESOURCE(ENQ-CRUD-ACTIVE)
          LENGTH(LENGTH OF  ENQ-CRUD-ACTIVE)
          NOHANDLE
          NOSUSPEND
          TASK
     END-EXEC.

     IF  EIBRESP EQUAL DFHRESP(ENQBUSY)
         PERFORM 9000-RETURN THRU 9000-EXIT.

     PERFORM UNTIL LAST-MATCHING-PO
         PERFORM 1100-ENQ-CRUD       THRU 1100-EXIT
         PERFORM 1200-READNEXT-PO    THRU 1200-EXIT
         PERFORM 1300-APPLY-CRUD     THRU 1300-EXIT
         PERFORM 1400-DELETE-PO      THRU 1400-EXIT
         PERFORM 1500-SYNCPOINT      THRU 1500-EXIT
     END-PERFORM.

 1000-EXIT.
     EXIT.

*****************************************************************
** Obtain lock on CRUD level when necessary.                    **
*****************************************************************
 1100-ENQ-CRUD.
     IF  SUBTASK-LEVEL-ENQ NOT EQUAL 'Y'
         EXEC CICS ENQ RESOURCE(ENQ-CRUD)
              LENGTH(LENGTH OF  ENQ-CRUD)
              NOHANDLE
              NOSUSPEND
          TASK
       END-EXEC.

    IF  EIBRESP EQUAL DFHRESP(NORMAL)
        MOVE 'Y' TO CRUD-LEVEL-ENQ.

 1100-EXIT.
    EXIT.
```

Example 4-3 shows the dispatcher perspective.

Example 4-3 Dispatcher perspective

```
*****************************************************************
** ENQ Child CRUD level                                        **
*****************************************************************
 01  ENQ-CRUD.
     02  FILLER             PIC  X(08) VALUE 'CICSGRS_'.
     02  FILLER             PIC  X(04) VALUE 'OMS_'.
     02  FILLER             PIC  X(05) VALUE 'CRUD_'.

*****************************************************************
** ENQ Child CRUD level to establish ownership of the Purchase **
** Order by a dispatcher across the Parallel Sysplex.          **
*****************************************************************
 4000-READ-CFDT.

     MOVE LENGTH OF PO-RECORD TO PO-RECORD-LENGTH.

     EXEC CICS READ FILE(CFDT-PO)
          INTO(PO-RECORD)
          LENGTH(PO-RECORD-LENGTH)
          RIDFLD(PO-Key)
          GTEQ
          NOHANDLE
     END-EXEC.


     EXEC CICS ENQ RESOURCE(ENQ-CRUD)
          LENGTH(LENGTH OF  ENQ-CRUD)
          NOSUSPEND
          UOW
          NOHANDLE
     END-EXEC

     IF  EIBRESP EQUAL DFHRESP(NORMAL)
         MOVE PO-KEY     TO CRUD-PO
         PERFORM 5000-START-CRUD    THRU 5000-EXIT.

 4000-EXIT.
     EXIT.
```

4.3.3 Enqueue list

An ECBLIST has traditionally been used to manage parallel processing of tasks within an application. However, this approach is limited to processing within a single CICS region. A similar capability can be achieved for sysplex-wide multi-region activity by using GRS.

Scenarios that are related to managing sysplex-wide multi-region tasks are covered earlier in this chapter and in other chapters in this publication. These situations represent a pattern where an "event notification" mechanism is useful.

If a primary task is initiating a set of secondary tasks to perform an activity, an associative resource can be established to represent each of the secondary task invocations. Secondary tasks can then issue status updates to these resources for status changes, thereby effectively emitting an event that the primary tasks can respond to. See Figure 4-3.

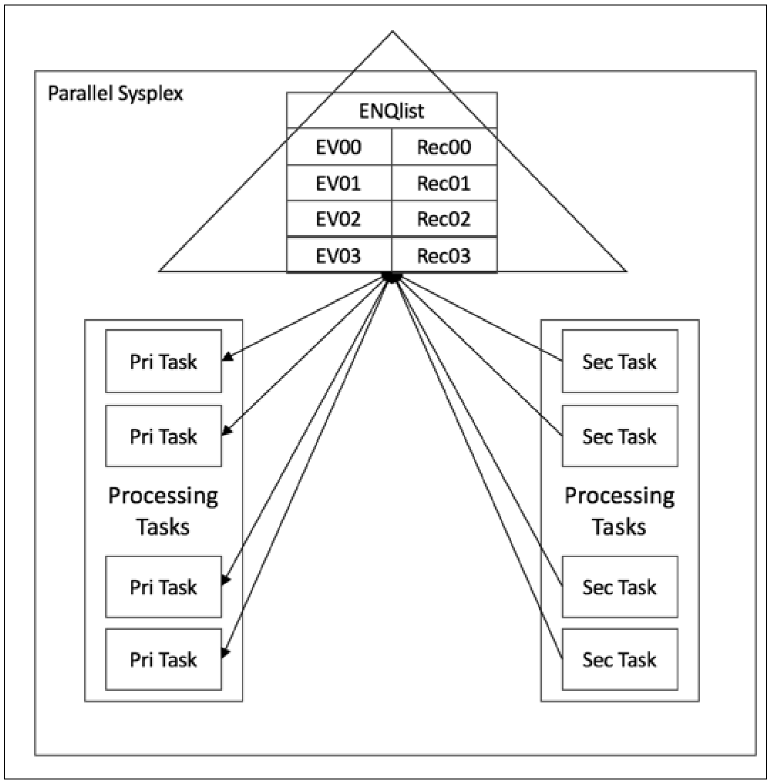


Figure 4-3 ENQ list

The following code samples demonstrate the actions required to utilize the "ENQ List" functionality. Example 4-4 shows a primary task checking the list.

Example 4-4 Primary task checking the list

```
PERFORM 0500-ENQ-PROCESS     THRU 0500-EXIT
         WITH TEST AFTER
         VARYING ENQ-INDEX     FROM 1 BY 1 UNTIL
                 ENQ-INDEX     = StoreRequest-num.

*****************************************************************
* ENQ on each TSQ name, as these are used by the Subtask        *
* to serialize the request.                                     *
*****************************************************************
 0500-ENQ-PROCESS.

     MOVE TSQ-NAME(ENQ-INDEX) TO TSQ-QNAME.

     EXEC CICS ENQ RESOURCE(TSQ-QNAME)
          LENGTH(LENGTH OF TSQ-QNAME) NOHANDLE
     END-EXEC.
 0500-EXIT.
     EXIT.
```

Example 4-5 shows a secondary task ENQ on resource.

Example 4-5 Secondary task ENQ on resource

```
0100-ENQ-RESOURCE.                                                      00022812
     MOVE LENGTH OF RETRIEVE-DATA TO RETRIEVE-LENGTH.

     EXEC CICS RETRIEVE
          INTO  (RETRIEVE-DATA)
          LENGTH(RETRIEVE-LENGTH)
          NOHANDLE
     END-EXEC.

     MOVE RETRIEVE-DATA TO TSQ-NAME.

     EXEC CICS ENQ
          RESOURCE(TSQ-NAME)
          LENGTH  (RETRIEVE-LENGTH)
          TASK
          NOHANDLE
     END-EXEC.

     EXEC CICS ASSIGN
          APPLID(APPLID) SYSID(SYSID)
     END-EXEC.

     EXEC CICS ASKTIME ABSTIME(CF-ABS)
          NOHANDLE
     END-EXEC.

 0100-EXIT.
     EXIT.
```

Example 4-6 shows a secondary task DEQ the resource and allows the primary task to resume.

Example 4-6 Secondary task DEQ the resource

```
0800-DEQ-RESOURCE.
     EXEC CICS DEQ
          RESOURCE(TSQ-NAME)
          LENGTH  (RETRIEVE-LENGTH)
          NOHANDLE
     END-EXEC.
 0800-EXIT.
     EXIT.
```

Those who are particularly observant might notice references to other CF features in the code samples. Chapter 7, “Combined functional use” on page 57 includes additional information about how combining the use of multiple CF features can provide higher qualities of service.

5 Coupling facility data tables

Coupling facility data tables (CFDTs) provide a method of file data sharing, using CICS file control, without the need for a file owning region (FOR) and without the need for Virtual Storage Access Method (VSAM) record-level sharing (RLS) support. CICS CFDT support provides rapid sharing of working data within a sysplex, with update integrity.

The data is held in an IBM Coupling Facility (CF) in a table that is similar to a shared user-maintained data table. Because read access and write access have similar performance, this form of table is useful for scratch pad data. After it is defined, a CFDT can be loaded initially with data from a VSAM data set.

Note: Upon completion of the load, no automatic synchronization of data between the CFDT and VSAM data set occurs.

A CFDT is often confused with a user-maintained data table (UMT) and a CICS-maintained data table (CMT). Although CFDTs have characteristics in common with UMTs and CMTs, they use different technologies. CFDTs provide sysplex-wide access using the coupling facility, and UMTs and CMTs use data spaces and, therefore, access is limited to CICS regions running within the same LPAR.

This chapter provides an overview of CFDTs and includes the following topics:

- Common uses of CFDTs
- Setting up CFDTs
- Examples of CFDT use

5.1 Common uses of CFDTs

Typically, CFDTs are used for the following reasons:

- A scratch pad for in-memory objects
- Sharing of commonly-used dynamic data that must be consistent across regions

5.2 Setting up CFDTs

This section summarizes the steps that are required to set up a CFDT server and to update CICS to access the server. It also provides information about how to access the CFDT from an application.

You can find information about setting up and running a coupling facility data table server in IBM Knowledge Center.

5.2.1 Defining resources to the CF

Each CFDT pool requires a CF list structure to be defined. The definition is created by updating the CF Resource Manager (CFRM) policy using the IXCMIAPU utility. The name of the structure is `DFHCFLS_`*`poolname`*. In addition to the name of the structure, space information is required. IBM provides a tool to assist with the space estimation.

Figure 5-1 shows a sample structure definition, as defined in the CFRM policy, that can support a CFDT pool called `PRODCFT1`.

```
STRUCTURE NAME(DFHCFLS_PRODCFT1)
   SIZE(79872)
   INITSIZE(32768)
   PREFLIST(FACIL01,FACIL02)
```

Figure 5-1 Example of a CFDT structure definition

5.2.2 Creating the CFDT server

A CICS region can simultaneously access CFDTs in multiple pools. However, for each pool, a dedicated named counter server region is required on each LPAR where access is needed. The access and management of the pools is transparent to the CICS regions and applications that use them.

Too many developers don't realize they can expand beyond one region or one LPAR by using the CFDT in the Coupling Facility. —
Randy Frerking, Distinguished Systems Engineer, Walmart

Detailed instructions for setting up the server, including information about configuration and security are provided in IBM Knowledge Center.

Example 5-1 shows an example of a CFDT server JCL for the `PRODCFT1` pool.

Example 5-1 CFDT server sample JCL

```
//CICSCF1 JOB ...
//CFSERVER EXEC PGM=DFHCFMN,REGION=32M,TIME=NOLIMIT
//STEPLIB   DD  DISP=SHR,DSN=CICS.SDFHAUTH
//          DD  DISP=SHR,DSN=CICS.SDFHLIC
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD  *
POOLNAME=PRODCFT1
/*
```

5.2.3 Defining CFDTs in CICS

No CICS region system initialization changes are required to use CFDTs.

From a CICS perspective, a CFDT is defined as a FILE resource with attributes that indicate the data maps to a CF list structure. You can define multiple CFDTs within a CICS region and group individual CFDTs together into pools. CFDTs in the same pool share a CF list structure.

The following CICS FILE definition attributes cause CFDTs to be created:

- `TABLE(CF)`
- `CFDTPOOL(poolname)`

The following attributes affect the initial loading of a CFDT from a VSAM data set:

- `LOAD(YES|NO)`
- `DSNAME()`

Review additional attributes, such as `KEYLENGTH`, `MAXNUMRECS`, `RECOVERY`, `TABLENAME`, and `UPDATEMODEL` and set them appropriately to meet the requirements of the application.

Note: The maximum key length for a CFDT is 16 characters.

5.2.4 Accessing CFDTs

An application that wants to use a CFDT defines the resource as outlined in the previous section. Access to the CFDT is performed using the same CICS API commands that are used to access VSAM files.

Note: The longevity of the data in a CFDT is not dependent upon the application, CICS regions, or CFDT servers that access them. When allocated, the CFDT data in the CF list structure remains until such time as the application deletes the data or until a CF structure failure results in loss of the CF list structure.

To display the CFDT pools that are currently defined to a CICS region, issue the following command:

```
CEMT INQUIRE CFDTPOOL
```

A user can also display the CFDTs that are currently defined to a CICS region. For example, the following command displays all files that are defined to the CFDT `PRODCFT1` pool:

```
CEMT INQUIRE FILE CFDTPOOL(PRODCFT1)
```

5.3 Examples of CFDT use

The CFDT provides a centralized “scratch pad” that is available to multiple regions throughout a sysplex and is accessible through the same methods as other data stores. This capability can be useful in a number of scenarios. However, you need to consider some restrictions that are associated with the CFDT. In particular, keep in mind the key length restriction of 16 bytes for a CFDT.

The following examples explore ways that you can use the CFDT beneficially, with the assumption that the data formats that are associated with the workload are within the confines of the stated CFDT restrictions.

The samples in this chapter assume familiarity with other use cases and examples within this book. Be sure to review Chapter 7, “Combined functional use” on page 57 to see how these various technologies are combined to achieve fully-realized solutions.

5.3.1 Caching information

Caching is a technique that is used throughout information processing, from the hardware level to the highest-level application layers. It is a common solution to avoiding long-path requests, such as disk I/O or network calls, for regularly accessed data. The CFDT can be used as a caching mechanism for some types of workloads.

For tasks that retrieve information from remote sources, such as remote hosts on a network, and where that information might potentially be requested repeatedly, a cache can be beneficial. In this case, you can set up a CICS program to use CFDT as that cache.

The basic logic associated with this approach is similar to any other caching implementation. The process checks cache first for a requested value, and then returns the value on a cache hit. Otherwise, it requests the information from the remote source, adds the value to the cache, and then returns the result (as illustrated in Figure 5-2 on page 35).

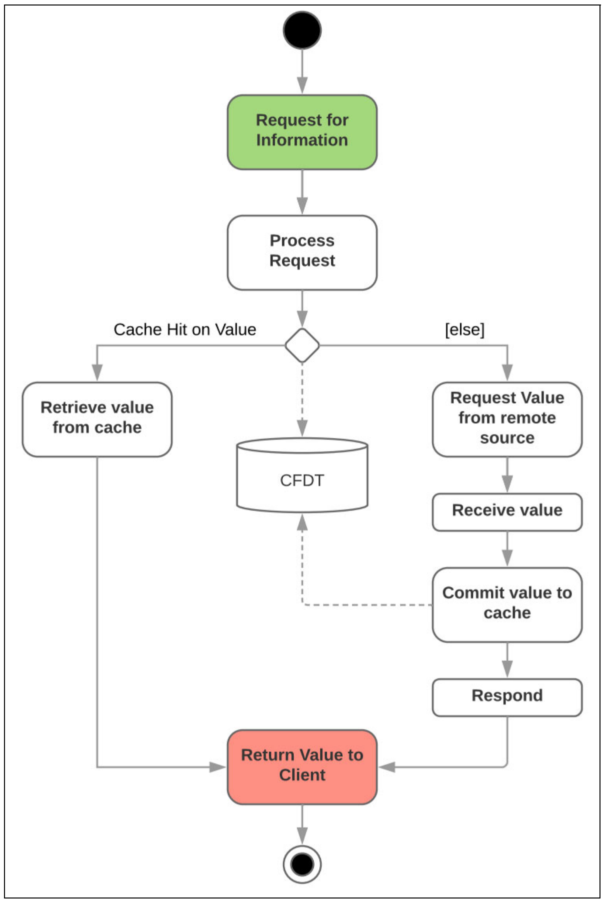


Figure 5-2 Basic caching logic

As described in 5.2.3, “Defining CFDTs in CICS” on page 33, establishing the CFDT FILE entry is accomplished with some basic Resource Definition Online (RDO) specifications. With the CFDT defined, the relevant parts of this process come down to checking the CFDT cache upon a request and potentially updating the CFDT cache in the event of a cache miss. The following code samples provide examples of these actions.

Example 5-2 shows working storage specifications for sample cache entries.

Example 5-2 Cache resources

```
*****************************************************************
* Cache resources.                                              *
*****************************************************************
 01  CF-RESPONSE           PIC S9(08) COMP  VALUE ZERO.
 01  CF-LENGTH             PIC S9(04) COMP  VALUE ZERO.

 01  CF-FILE               PIC  X(08) VALUE 'STORE$CF'.

 01  CF-RECORD.
     02  CF-KEY.
         05  CF-STORE      PIC  9(05).
         05  CF-UPC        PIC  9(10).
     02  CF-ABS            PIC S9(15) COMP-3.
     02  CF-PRICE          PIC  9(06).
     02  CF-ONHAND         PIC  9(04).
     02  CF-DESC           PIC  X(65).
```

Example 5-3 shows check cache for a requested value.

Example 5-3 Check cache

```
*****************************************************************
* Check Cache (CFDT) for Store and UPC information.             *
*****************************************************************
 0300-CHECK-CACHE.
     MOVE TSQ-REQ-STORE       TO CF-STORE.
     MOVE TSQ-REQ-UPC         TO CF-UPC.

     EXEC CICS READ
          FILE  (CF-FILE)
          INTO  (CF-RECORD)
          RIDFLD(CF-KEY)
          LENGTH(CF-LENGTH)
          RESP  (CF-RESPONSE)
          NOHANDLE
     END-EXEC.

     IF  CF-RESPONSE EQUAL DFHRESP(NORMAL)
         MOVE CF-STORE        TO TSQ-RESP-STORE
         MOVE CF-UPC          TO TSQ-RESP-UPC
         MOVE CF-PRICE        TO TSQ-RESP-PRICE
         MOVE CF-ONHAND       TO TSQ-RESP-ONHAND
         MOVE CF-DESC         TO TSQ-RESP-DESC.

 0300-EXIT.
      EXIT.
```

Example 5-4 shows the update to the CFDT cache upon retrieval of the information from the remote source.

Example 5-4 Write cache

```
0650-WRITE-CACHE.
     MOVE TSQ-RESP-STORE         TO CF-STORE.
     MOVE TSQ-RESP-UPC           TO CF-UPC.
     MOVE TSQ-RESP-PRICE         TO CF-PRICE.
     MOVE TSQ-RESP-ONHAND        TO CF-ONHAND.
     MOVE TSQ-RESP-DESC          TO CF-DESC.

     EXEC CICS WRITE
          FILE  (CF-FILE)
          FROM  (CF-RECORD)
          RIDFLD(CF-KEY)
          LENGTH(CF-LENGTH)
          RESP  (CF-RESPONSE)
          NOHANDLE
     END-EXEC.

 0650-EXIT.
     EXIT.
```

You need to consider other factors for a caching implementation such as that described here. For example, cache entry invalidation is sometimes performed for records with a least recently used (LRU) algorithm. One approach might include a classification of entries based on activity. You might use a weighting scale that ranges from hot (high activity) to cold (little or no activity) to rank the cache entries, allowing for a precise invalidation process. Alternatively, a simple time-based invalidation process can be employed.

The caching examples in this book generally employ a time-based expiration process, which involves a time to live (TTL) value and a time stamp that is logged for each entry in the cache. A subsequent process then calculates the delta of these values, compares that change to current time, and expires the appropriate records.

Example 5-5 shows sample code that uses the `ABSTIME` time stamp with a predefined TTL (specified as 1,800 seconds for a 30-minute TTL). This example stores them with a piece of data.

Example 5-5 Example 1

```
*****************************************************************
* DEFINE LOCAL VARIABLES                                        *
*****************************************************************
 01  CURRENT-ABS           PIC S9(15) COMP-3 VALUE ZEROES.
 01  THIRTY-MINUTES        PIC S9(15) COMP-3 VALUE 1800.
 01  ONE                   PIC S9(08) COMP   VALUE ZEROES.

 01  DS-RECORD.
     02  DS-KEY-16.
         05  DS-KEY.
           10 DS-KEY-STCKE  PIC  X(06) VALUE LOW-VALUES.
           10 DS-KEY-NC     PIC  X(02) VALUE LOW-VALUES.
         05  DS-SEGMENT     PIC  9(04) VALUE ZEROES COMP.
         05  DS-PREFIX      PIC  9(04) VALUE ZEROES COMP.
         05  DS-SUFFIX      PIC  9(08) VALUE ZEROES COMP.
     02  DS-ABS             PIC S9(15) VALUE ZEROES COMP-3.
     02  DS-TTL             PIC S9(07) VALUE ZEROES COMP-3.
     02  DS-SEGMENTS        PIC  9(04) VALUE ZEROES COMP.
     02  FILLER             PIC  X(15).
     02  DS-KS-KEY          PIC  X(255).
     02  DS-MEDIA-TYPE      PIC  X(56).
     02  DS-DATA            PIC  X(32000).

*****************************************************************
* Initialize File Control names.                                *
* Get absolute time for the DATA store.                         *
*****************************************************************
 1000-INITIALIZE.
     EXEC CICS ASKTIME ABSTIME(CURRENT-ABS) NOHANDLE
     END-EXEC.

     MOVE EIBTRNID                    TO DATA-STORE.
     MOVE EIBTRNID                    TO KEY-STORE.

 1000-EXIT.
     EXIT.

*****************************************************************
* Write cache DATA store record.                                *
* Set TTL to 30 minutes, which is expressed in seconds at 1800. *
*****************************************************************
 2000-WRITE-DATA.
```

```
    MOVE LENGTH OF DS-RECORD          TO DS-LENGTH.
    MOVE CURRENT-ABS                  TO DS-ABS.
    MOVE THIRTY-MINUTES               TO DS-TTL.

    EXEC CICS WRITE
         FILE  (DATA-STORE)
         RIDFLD(DS-KEY-16)
         FROM  (DS-RECORD)
         LENGTH(DS-LENGTH)
         NOHANDLE
    END-EXEC.

 2000-EXIT.
    EXIT.
```

Example 5-6 provides an example of comparing a current time stamp with the TTL of the entry and then deleting it accordingly.

Example 5-6 Example 2

```
*****************************************************************
* DEFINE LOCAL VARIABLES                                        *
*****************************************************************
 01  CURRENT-ABS           PIC S9(15) COMP-3 VALUE ZEROES.
 01  RELATIVE-TIME         PIC S9(15) COMP-3 VALUE ZEROES.
 01  ONE                   PIC S9(08) COMP   VALUE ZEROES.
 01  EOF                   PIC  X(01) VALUE SPACES.

 01  TTL-MILLISECONDS      PIC S9(15) VALUE ZEROES COMP-3.
 01  FILLER.
     02  TTL-SEC-MS.
         03  TTL-SECONDS    PIC  9(06) VALUE ZEROES.
         03  FILLER         PIC  9(03) VALUE ZEROES.
     02  FILLER REDEFINES TTL-SEC-MS.
         03  TTL-TIME       PIC  9(09).

 01  DS-RECORD.
     02  DS-KEY-16.
         05  DS-KEY.
           10 DS-KEY-STCKE  PIC  X(06) VALUE LOW-VALUES.
           10 DS-KEY-NC     PIC  X(02) VALUE LOW-VALUES.
         05  DS-SEGMENT     PIC  9(04) VALUE ZEROES COMP.
         05  DS-PREFIX      PIC  9(04) VALUE ZEROES COMP.
         05  DS-SUFFIX      PIC  9(08) VALUE ZEROES COMP.
     02  DS-ABS             PIC S9(15) VALUE ZEROES COMP-3.
     02  DS-TTL             PIC S9(07) VALUE ZEROES COMP-3.
     02  DS-SEGMENTS        PIC  9(04) VALUE ZEROES COMP.
     02  FILLER             PIC  X(15).
     02  DS-KS-KEY          PIC  X(255).
     02  DS-MEDIA-TYPE      PIC  X(56).
     02  DS-DATA            PIC  X(32000).

*****************************************************************
* Read cache DATA store record.                                 *
* Since there can be multiple segments for a single cache       *
* record, only check the first record for expiration.           *
```

```
*****************************************************************
 2000-READ-DATA.
     MOVE LENGTH OF DS-RECORD          TO DS-LENGTH.

     EXEC CICS READ
          FILE  (DATA-STORE)
          RIDFLD(DS-KEY-16)
          INTO  (DS-RECORD)
          LENGTH(DS-LENGTH)
          GTEQ
          NOHANDLE
     END-EXEC.

     MOVE DS-TTL                       TO TTL-SECONDS.
     MOVE TTL-TIME                     TO TTL-MILLISECONDS.

     SUBTRACT DS-ABS FROM CURRENT-ABS GIVING RELATIVE-TIME.
     IF  RELATIVE-TIME GREATER THAN TTL-MILLISECONDS
         PERFORM 3000-DELETE-CACHE   THRU 3000-EXIT.

     ADD ONE                           TO DS-SUFFIX.

 2000-EXIT.
     EXIT.
```

5.3.2 Logging work items

When running multiple parallel processes to perform certain work, it can be beneficial to have a central location for those processes to acquire work items from. This concept is similar to a to-do list that multiple folks are working on or the backlog for a scrum team, where multiple developers are picking up features to work on. The CFDT provides a great mechanism for this type of concept.

Many examples in this publication involve parallel processing of tasks that are working on a common set of requests. These requests can be recorded independently and then processed by downstream tasks. By using the CFDT, the requests can be logged to a central location and then processed asynchronously by other tasks. To ensure uniqueness for work items, a sequence number can be appended to each item. See Figure 5-3 on page 40.

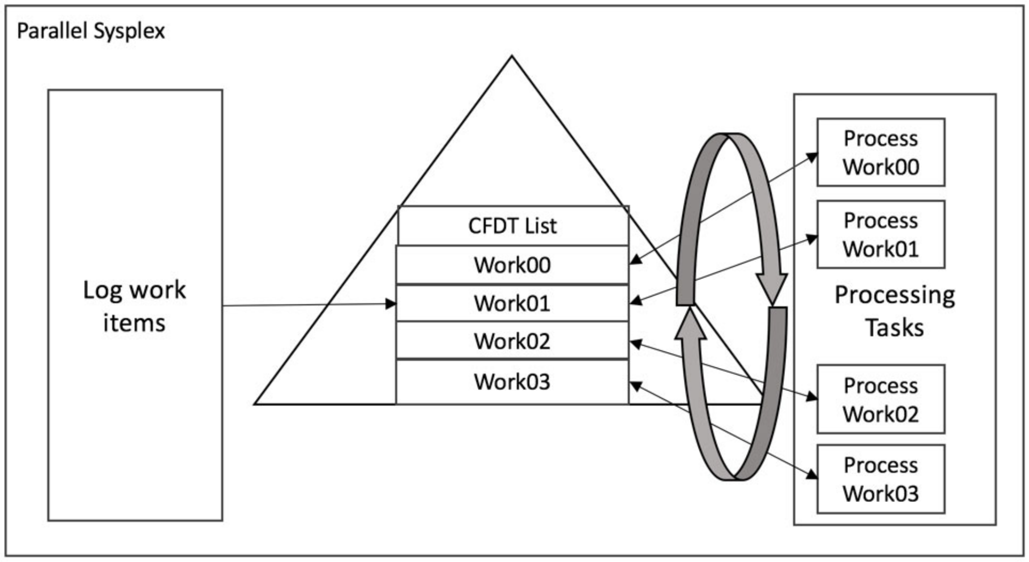
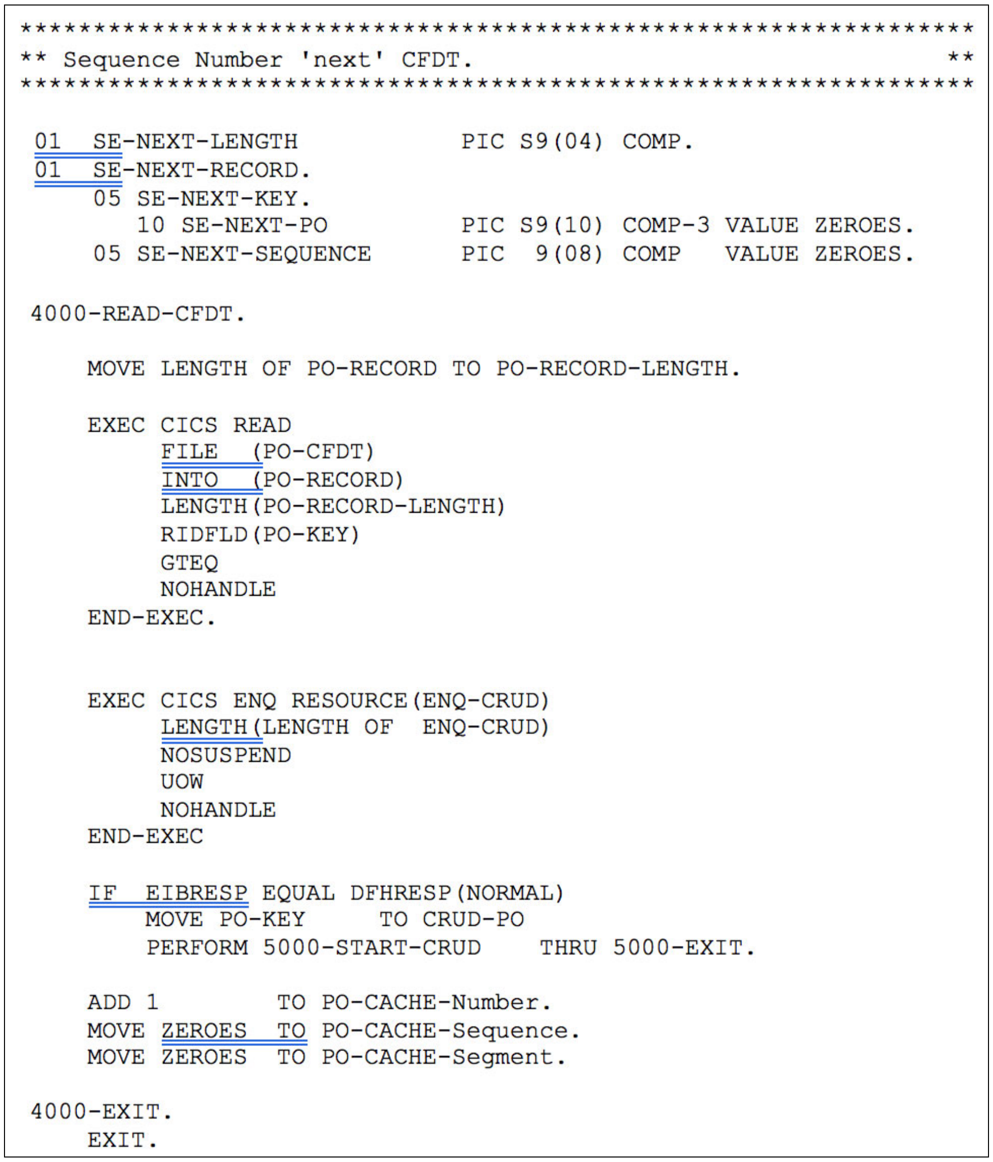


Figure 5-3 A centralized work item list

Although you need to consider many other factors for controlling parallel processing, the general mechanics of this approach are demonstrated (within the context of purchase order processing) in the code sample shown in Figure 5-4.



```
*****************************************************************
** Sequence Number 'next' CFDT.                                **
*****************************************************************

 01  SE-NEXT-LENGTH            PIC S9(04) COMP.
 01  SE-NEXT-RECORD.
     05 SE-NEXT-KEY.
        10 SE-NEXT-PO          PIC S9(10) COMP-3 VALUE ZEROES.
     05 SE-NEXT-SEQUENCE       PIC  9(08) COMP   VALUE ZEROES.

 4000-READ-CFDT.

     MOVE LENGTH OF PO-RECORD TO PO-RECORD-LENGTH.

     EXEC CICS READ
          FILE   (PO-CFDT)
          INTO   (PO-RECORD)
          LENGTH(PO-RECORD-LENGTH)
          RIDFLD(PO-KEY)
          GTEQ
          NOHANDLE
     END-EXEC.


     EXEC CICS ENQ RESOURCE(ENQ-CRUD)
          LENGTH(LENGTH OF  ENQ-CRUD)
          NOSUSPEND
          UOW
          NOHANDLE
     END-EXEC

     IF  EIBRESP EQUAL DFHRESP(NORMAL)
         MOVE PO-KEY     TO CRUD-PO
         PERFORM 5000-START-CRUD    THRU 5000-EXIT.

     ADD 1          TO PO-CACHE-Number.
     MOVE ZEROES   TO PO-CACHE-Sequence.
     MOVE ZEROES   TO PO-CACHE-Segment.

 4000-EXIT.
     EXIT.
```

Figure 5-4 Purchase order processing

5.3.3 Flexibility for data stores

Different types of workloads warrant differing design decisions. Access patterns that are heavily weighted towards write activity over read activity or items that are simply accessed at an extremely high rate are examples where a faster medium than disk might be considered. In situations like this, the CFDT is a compelling option for data storage over general disk-backed data stores.

CFDT provides a significant amount of flexibility in this context. The CICS API commands to access a CFDT are the same commands as for any other VSAM-type file that is managed by CICS, such as shared data tables, local VSAM, remote VSAM via an FOR, or VSAM RLS.

The type of file that is accessed by these APIs is determined by the CICS RDO parameters and not by the application code.

The ability to control the data storage location via RDO greatly reduces complexity within application code and gives system administrators flexibility in how workloads are managed. As an example, consider a situation where a process that typically uses VSAM RLS is expected to be primarily write heavy. To reduce the penalty of longer response times for all the write activity to disk, the data store can be defined in CFDT and take advantage of the faster path of InfiniBand to memory in the CF rather than I/O over Fibre Channel connection (FICON) to disk arrays. And this is done by a simple RDO change, with no alteration to the application code itself (Figure 5-5).

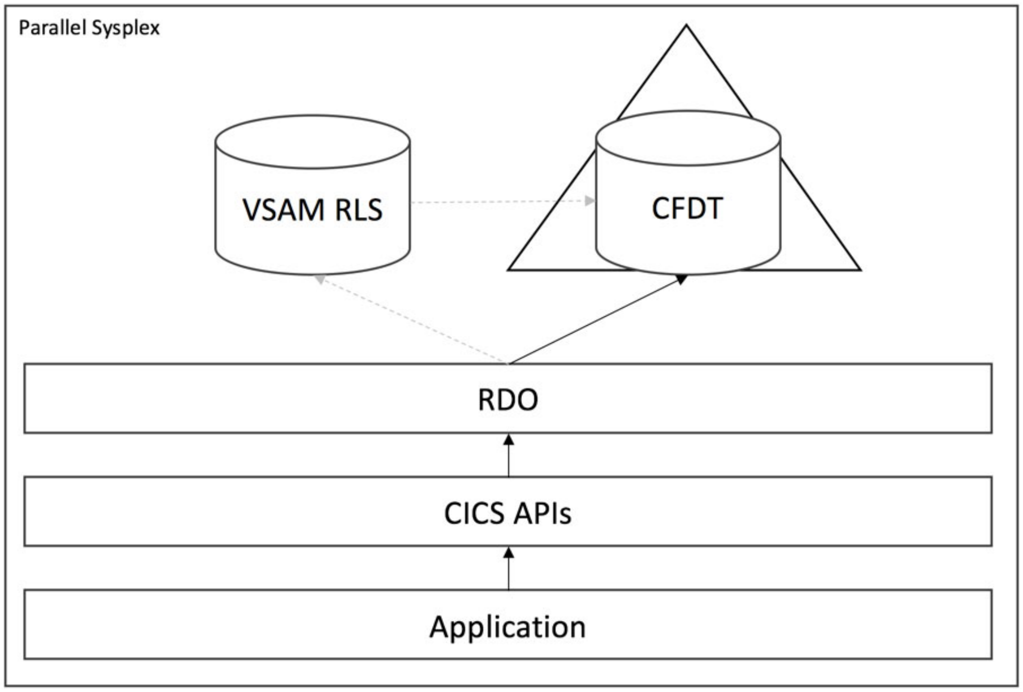


Figure 5-5 Logical view of FILE processing

This method is achieved by adjusting the RDO parameters, as described in 5.2.3, “Defining CFDTs in CICS” on page 33. These simple changes can allow the system to determine an appropriate data store location transparently to the application by specifying `TABLE(CF)` and by providing a `POOLNAME()` value in the resource defintion.

6

Shared temporary storage

Temporary storage (TS) is the primary CICS facility for storing data that must be available to multiple transactions. Data items in temporary storage are kept in temporary storage queues. The items can be retrieved by the originating task, or by any other task, by using the symbolic name assigned to the temporary storage queue.

Temporary storage resides in one of three places:

- Locally in Memory referred to as TS main
- Locally in a VSAM data set referred to as TS Auxiliary
- Centrally within a CF list structure referred to as Shared TS

The choice as to which type of TS to use depends upon factors, such as longevity of the data, recovery, and access.

You can find a more detailed overview about CICS temporary storage queues in IBM Knowledge Center.

This chapter concentrates on shared temporary storage, which is accessible by CICS regions anywhere within a sysplex. Although you can set up special CICS regions to centrally manage local TS and to allow wider access by CICS regions using function shipping, shared TS is a more versatile and highly-available solution for CICS regions that are running in a parallel sysplex environment.

This chapter incudes the following topics:

- Common uses of temporary storage
- Setting up shared temporary storage
- Use cases for shared temporary storage

6.1 Common uses of temporary storage

Shared TS is frequently used to share data between applications that are hosted within multiple CICS regions in a sysplex. TS includes the following common uses:

- Shopping baskets
- History of actions performed by a task or series of tasks
- Synchronization of asynchronous activities within one or more CICS regions

Local TS (main and auxiliary) is often migrated to shared TS to remove application to system affinities, which might otherwise tie a sequence of tasks that share TS data to a single CICS region.

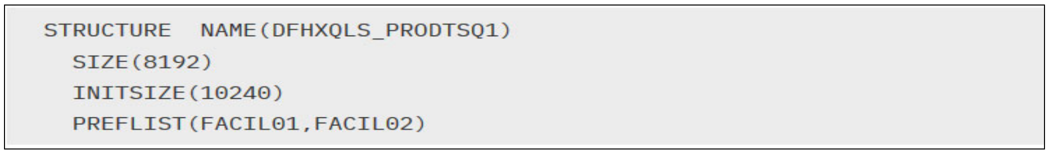
6.2 Setting up shared temporary storage

This section summarizes the steps that are required to set up a shared TS server and to update CICS to access that server. You can find full details about setting up and running a temporary storage server in IBM Knowledge Center.

6.2.1 Defining resources to the CF

Each shared TS pool requires a defined CF list structure. The definition is created by updating the coupling facility resource management (CFRM) policy using the IXCMIAPU utility. The name of the structure is `DFHXQLS_`*`poolname`*. In addition to the name of the structure, space information is required. IBM provides a tool to assist with the space estimation.

Figure 6-1 shows a sample structure definition, as defined in the CFRM policy, that supports a shared TS pool called `PRODTSQ1`.



```
STRUCTURE  NAME(DFHXQLS_PRODTSQ1)
  SIZE(8192)
  INITSIZE(10240)
  PREFLIST(FACIL01,FACIL02)
```

Figure 6-1 Example of CICS shared TS structure definition

6.2.2 Creating the shared TS server

A CICS region can simultaneously access shared TS queues in multiple pools. However, for each pool, a dedicated shared TS server is required on each LPAR where access is required. You can find detailed instructions about setting up a temporary storage server, including information about configuration and security, in IBM Knowledge Center.

Figure 6-2 shows an example of the shared TS server JCL for the PRODTSQ1 pool.

```
//CICSTS1 JOB …
//TSSERVER EXEC PGM=DFHXQMN,REGION=32M,TIME=NOLIMIT
//STEPLIB   DD  DISP=SHR,DSN=CICS.SDFHAUTH
//          DD  DISP=SHR,DSN=CICS.SDFHLIC
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD  *
POOLNAME=PRODTSQ1
/*
```

Figure 6-2 Shared TS server sample JCL

6.2.3 Changes to CICS to access the shared TS server

A TS queue is created when it is first written to using the **WRITEQ TS** API command. For local queues, the location can be specified as MAIN or AUXILIARY, with the location defaulting to auxiliary.

Optionally, administrators can define a TSMODEL, which allows the location, recovery, and automatic deletion values of a queue to be specified. Typically, a TSMODEL is set up so that queues with a certain prefix, for example an application name, have a common location and characteristics. Shared TS queues require a TSMODEL so that the POOL name can be specified.

The following sample definition causes all queues with the REDB prefix to be in a shared TS pool called PRODTSQ1, and queues that are not accessed for 30 minutes are automatically deleted:

```
DEFINE TSMODEL(BOOKS) GR(REDBOOK) PREFIX(REDB) POOL(PRODTSQ1) EXPIRYINTM(30)
```

To display the currently defined TSMODELs, including those that map to shared TS pools, issue the following command:

```
CEMT INQUIRE TSMODEL
```

To display the currently defined local TS queues in a CICS region, issue the following command:

```
CEMT INQUIRE TSQUEUE
```

By default, shared TS queues are not returned. To display TS queues that reside in a shared TS pool, specify the POOLNAME attribute. For example, the following command returns all TS queues in the POOLTSQ1 pool:

```
CEMT INQUIRE TSQUEUE POOLNAME(POOLTSQ1)
```

6.3 Use cases for shared temporary storage

A lot of flexibility is provided by a shared data location, where multiple transactions across multiple regions can access certain assets. This section describes some real-world use cases of using shared TS queues.

6.3.1 Adjusting priority based on HTTP method

You can find code samples for using a CONVERTER on the URIMAP to extract the HTTP method from incoming HTTP requests in 3.3.4, “Tracking activity” on page 17. In that scenario, the objective was to track the distribution of incoming HTTP methods. However, this capability can also be used in other ways. For example, an excellent feature of the CONVERTER program is that pre- or post-processing can be invoked. Using this capability, you can apply pre-processing logic to an incoming request after its HTTP method has been identified.

Within CICS, the TRANSACTION definition allows you to set a single PRIORITY for a given transaction. Thus, for a transaction that is associated with an HTTP service request, all requests are given the same priority, regardless of HTTP method.

There are instances where it is advantageous to set the transaction priority based on the HTTP method. Using shared TS and the CONVERTER program it is possible to perform this adjustment dynamically.

The following mechanisms are used to achieve this capability:

- A partitioned data set (PDS) member that defines the priorities

  The PDS member contains a priority value that is assigned to each HTTP method and serves as a persistent record of these values. It is made available to CICS through a DOCUMENT TEMPLATE.
- Centralizing the information using a shared TS

  By loading this information into a shared TS queue, all associated regions in the sysplex have access to the values from a single shared location. Any change to the values in the shared TS queue are available immediately to all associated regions in the sysplex, without needing to issue a NEW COPY for the DOC TEMPLATE in each of those CICS regions.
- A CONVERTER to adjust the priority of the task

  The CONVERTER extracts the HTTP method of the incoming request, consults the shared TS to acquire the specified method-based prioritization values, and adjusts the PRIORITY of the task accordingly for this individual request.

The persistent record of the HTTP method priority assignments is defined in a PDS member as follows:

```
PRIORITY POST=0100,GET=0200,PUT=0100,DELETE=0050
```

The code sample shown in Example 6-1 reflects the steps for acquiring the information from the PDS and loading it to a shared TS queue.

Example 6-1 Acquiring and sharing information

```
*****************************************************************
* HTTP METHOD priority resources.                               *
*****************************************************************
 01  TS-LENGTH              PIC S9(04) COMP VALUE ZERO.
 01  TS-ITEM                 PIC S9(04) COMP VALUE 1.

 01  TS-PRIORITY-QUEUE.
     02  FILLER              PIC  X(08) VALUE 'PRIORITY'.
     02  FILLER              PIC  X(01) VALUE '_'.
     02  TS-TRANID           PIC  X(04) VALUE SPACES.
     02  FILLER              PIC  X(03) VALUE SPACES.

 01  TS-PRIORITY-DATA.
     05  PRIORITY-POST       PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-GET        PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-PUT        PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-DELETE     PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-HEAD       PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-01    PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-02    PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-03    PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-04    PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-05    PIC S9(08) COMP VALUE ZERO.

*****************************************************************
* Document Template to obtain the transaction priority PDS.     *
*****************************************************************
 01  TEMPLATE-TOKEN         PIC  X(16) VALUE SPACES.
 01  TEMPLATE-LENGTH        PIC S9(08) COMP VALUE ZEROES.

 01  TEMPLATE-PDS-NAME.
     02  TEMPLATE-TRANID    PIC  X(04).
     02  FILLER             PIC  X(44) VALUE SPACES.

 01  TEMPLATE-PRIORITY.
     02  FILLER             PIC  X(14) VALUE 'PRIORITY POST='.
     02  TEMPLATE-POST      PIC  9(04).
     02  FILLER             PIC  X(05) VALUE ',GET='.
     02  TEMPLATE-GET       PIC  9(04).
     02  FILLER             PIC  X(05) VALUE ',PUT='.
     02  TEMPLATE-PUT       PIC  9(04).
     02  FILLER             PIC  X(08) VALUE ',DELETE='.
     02  TEMPLATE-DELETE    PIC  9(04).
     02  FILLER             PIC  X(06) VALUE ',HEAD='.
     02  TEMPLATE-HEAD      PIC  9(04).

*****************************************************************
* Get Transaction Priority for a CICS HTTP service from PDS.    *
*****************************************************************
 0100-DOCTEMPLATE.
     EXEC CICS DOCUMENT CREATE
          DOCTOKEN(TEMPLATE-TOKEN)
          TEMPLATE(TEMPLATE-PDS-NAME)
          NOHANDLE
     END-EXEC.

     MOVE LENGTH OF TEMPLATE-PRIORITY  TO TEMPLATE-LENGTH.

     IF  EIBRESP EQUAL DFHRESP(NORMAL)
```

```
         EXEC CICS DOCUMENT RETRIEVE DOCTOKEN(TEMPLATE-TOKEN)
              INTO     (TEMPLATE-PRIORITY)
              LENGTH   (TEMPLATE-LENGTH)
              MAXLENGTH(TEMPLATE-LENGTH)
              DATAONLY
              NOHANDLE
         END-EXEC.

 0100-EXIT.
     EXIT.

*****************************************************************
* Write Transaction Priority list to TS Server/CF.              *
*****************************************************************
 0200-WRITE-TSQ.
     MOVE LENGTH OF TS-PRIORITY-DATA   TO TS-LENGTH.
     MOVE EIBTRNID                     TO TS-TRANID.

     MOVE TEMPLATE-POST                TO PRIORITY-POST.
     MOVE TEMPLATE-GET                 TO PRIORITY-GET.
     MOVE TEMPLATE-PUT                 TO PRIORITY-PUT.
     MOVE TEMPLATE-DELETE              TO PRIORITY-DELETE.
     MOVE TEMPLATE-HEAD                TO PRIORITY-HEAD.

     EXEC CICS WRITEQ
          QNAME (TS-PRIORITY-QUEUE)
          FROM  (TS-PRIORITY-DATA)
          LENGTH(TS-LENGTH)
          ITEM  (TS-ITEM)
          NOHANDLE
     END-EXEC.

 0200-EXIT.
     EXIT.
```

Then HTTP method priority can be ascertained by reading the shared TS queue, and the appropriate priority can be assigned by the CONVERTER to each task, as illustrated in Figure 6-3.

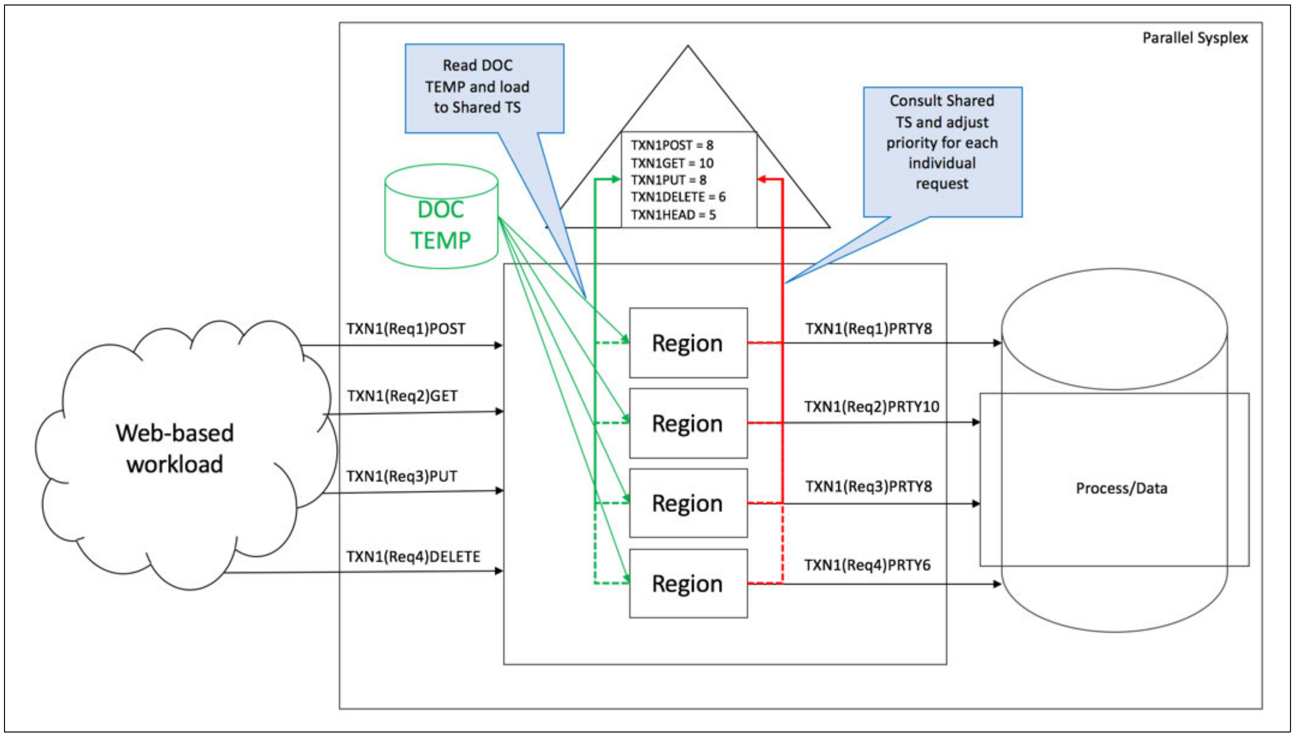


Figure 6-3 Adjust priority for each HTTP request

The code sample in Example 6-2 demonstrates defining the priority assignments and acquiring them from the shared TS queue.

Example 6-2 Priority assignments

```
*****************************************************************
* HTTP METHOD priority resources.                               *
*****************************************************************
 01  TS-LENGTH              PIC S9(04) COMP VALUE ZERO.
 01  TS-ITEM                PIC S9(04) COMP VALUE 1.

 01  TS-PRIORITY-QUEUE.
     02  FILLER             PIC  X(08) VALUE 'PRIORITY'.
     02  FILLER             PIC  X(01) VALUE '_'.
     02  TS-TRANID          PIC  X(04) VALUE SPACES.
     02  FILLER             PIC  X(03) VALUE SPACES.

 01  TS-PRIORITY-DATA.
     05  PRIORITY-POST      PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-GET       PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-PUT       PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-DELETE    PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-HEAD      PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-01   PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-02   PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-03   PIC S9(08) COMP VALUE ZERO.
     05  PRIORITY-FILL-04   PIC S9(08) COMP VALUE ZERO.
```

```
    05  PRIORITY-FILL-05    PIC S9(08) COMP VALUE ZERO.

 01  ACTIVE-PRIORITY         PIC S9(08) COMP VALUE ZERO.

*****************************************************************
* Read Transaction Priority from TS Server/CF.                  *
*****************************************************************
 0100-READ-TSQ.
     MOVE LENGTH OF TS-PRIORITY-DATA   TO TS-LENGTH.
     MOVE EIBTRNID                     TO TS-TRANID.

     EXEC CICS READQ
          QNAME (TS-PRIORITY-QUEUE)
          INTO  (TS-PRIORITY-DATA)
          LENGTH(TS-LENGTH)
          ITEM  (1)
          NOHANDLE
     END-EXEC.

 0100-EXIT.
     EXIT.
```

Example 6-3 demonstrates adjusting the priority of a task based on the previously acquired values.

Example 6-3 Adjusting priority

```
*****************************************************************
* Set task priority by HTTP METHOD.                             *
*****************************************************************
 0400-SET-TASK-PRIORITY.
     EXEC CICS INQUIRE TRANSACTION(EIBTRNID)
         PRIORITY(ACTIVE-PRIORITY) NOHANDLE
     END-EXEC.

     IF  HTTP-METHOD   EQUAL HTTP-POST
         MOVE PRIORITY-POST      TO ACTIVE-PRIORITY.

     IF  HTTP-METHOD   EQUAL HTTP-GET
         MOVE PRIORITY-GET       TO ACTIVE-PRIORITY.

     IF  HTTP-METHOD   EQUAL HTTP-PUT
         MOVE PRIORITY-PUT       TO ACTIVE-PRIORITY.

     IF  HTTP-METHOD   EQUAL HTTP-DELETE
         MOVE PRIORITY-DELETE    TO ACTIVE-PRIORITY.

     IF  HTTP-METHOD   EQUAL HTTP-HEAD
         MOVE PRIORITY-HEAD      TO ACTIVE-PRIORITY.

     EXEC CICS CHANGE TASK PRIORITY(ACTIVE-PRIORITY)
          NOHANDLE
     END-EXEC.

 0400-EXIT.
     EXIT.
```

6.3.2 Sharing data among tasks

It can be beneficial in a number of scenarios to share data among different CICS tasks. A shared TS queue provides an efficient mechanism to achieve this purpose, particularly when related to sharing data among tasks in multiple regions that are spread across a Parallel Sysplex. The example in this section explores a scenario in which a CICS task delegates data processing work to other tasks that are running in CICS regions across the sysplex.

At a high level, this scenario has the following components:

- A client process sends an array of items that it is requesting information about to a primary service provider task.
- A collection of secondary tasks, which in turn become *clients*, pass web-service requests to multiple remote hosts that request information about individual items from the originating requests array.
- Several shared TS queues are used for sharing information among the different tasks that perform different parts of the workflow.

  The following types of shared TS queues are used in this scenario:

  – *Registry queue*: Used to keep track of status for requests and delegations.

  – *Request queue*: The location to drop individual request items for the secondary tasks to pick up and process.

  – *Response queue*: The location for secondary tasks to place responses that are received from remote hosts and for the primary task to get the response for delivery back to the client.

A simplified representation of the workflow for a request is depicted in Figure 6-4.

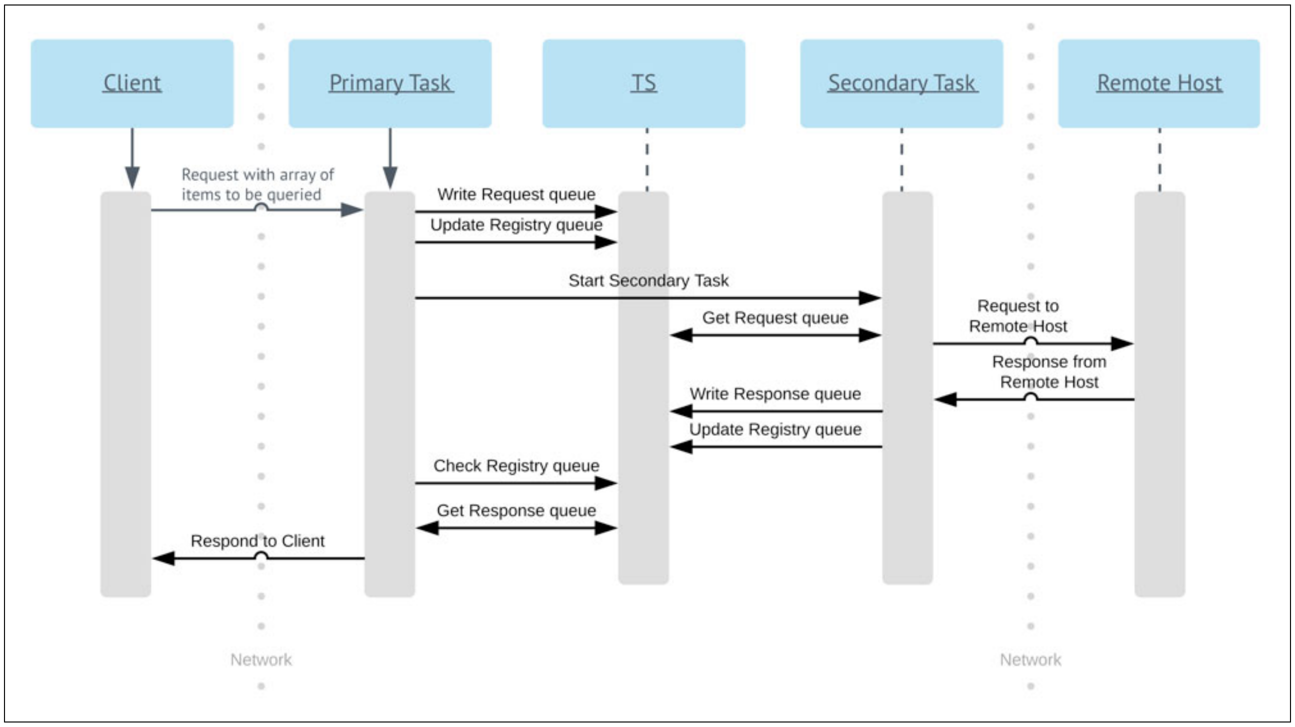
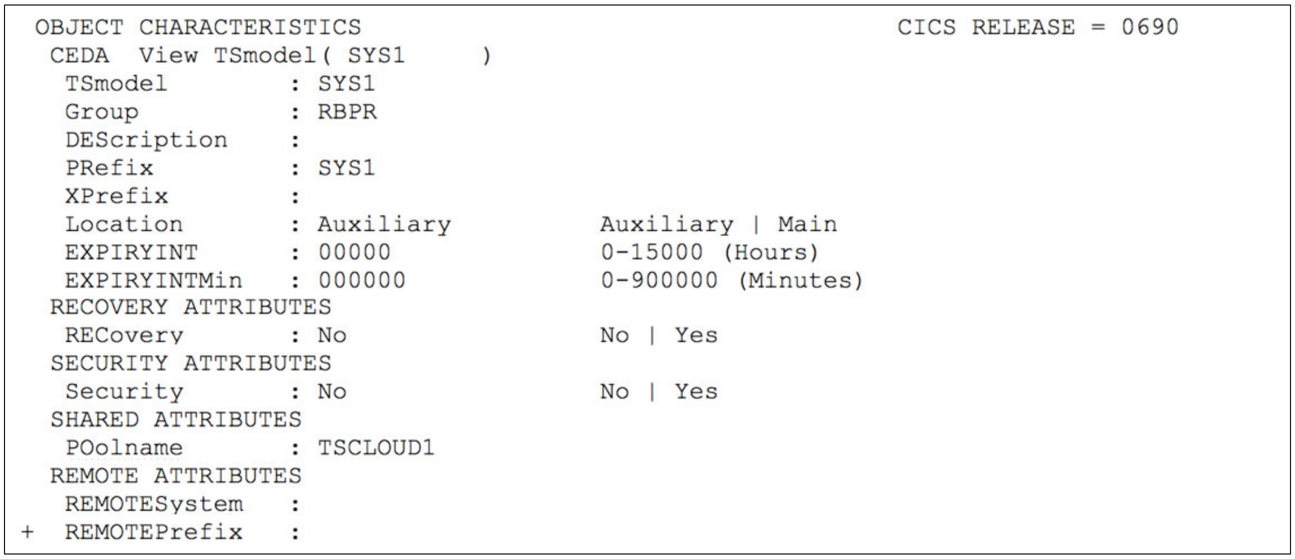


Figure 6-4 General workflow of request

The portion of the workflow in Figure 6-4 on page 51 between the *primary task* and the *remote host* is performed for each item in the original client request array, and the aggregated results are delivered to the client.

To sysplex enable this process, the TS queues must be defined as shared TS as described earlier, which is accomplished by defining a TSMODEL with the POOLNAME specified and associated with the appropriate TS server.

Figure 6-5 depicts a TSMODEL with a POOLNAME of `TSCLOUD1`.



```
 OBJECT CHARACTERISTICS                                    CICS RELEASE = 0690
  CEDA  View TSmodel( SYS1     )
   TSmodel        : SYS1
   Group          : RBPR
   DEScription    :
   PRefix         : SYS1
   XPrefix        :
   Location       : Auxiliary          Auxiliary | Main
   EXPIRYINT      : 00000              0-15000 (Hours)
   EXPIRYINTMin   : 000000             0-900000 (Minutes)
  RECOVERY ATTRIBUTES
   RECovery       : No                 No | Yes
  SECURITY ATTRIBUTES
   Security       : No                 No | Yes
  SHARED ATTRIBUTES
   POolname       : TSCLOUD1
  REMOTE ATTRIBUTES
   REMOTESystem   :
+  REMOTEPrefix   :
```

Figure 6-5 View TSMODEL

The programs associated with the primary and secondary tasks that define the TS queues must then use the appropriate PREFIX on the first four bytes for the queue name to result in the TSMODEL being used to define a shared TS queue.

Example 6-4 shows the language structures that demonstrate this concept from the perspective of the primary task.

Example 6-4 Language structures

```
*****************************************************************
* Registry resources.                                           *
*****************************************************************
 01  TSQ-REGISTRY.
     02  TSR-SYSID           PIC  X(04) VALUE SPACES.
     02  TSR-TASKN           PIC S9(07) VALUE ZEROS COMP-3.
     02  FILLER              PIC  X(08) VALUE 'REGISTRY'.

 01  TSQ-REGISTRY-DATA.
     02  TSR-DATA            PIC  X(10) VALUE 'NOTIFIED  '.

 01  ALL-TASKS-REGISTERED    PIC  X(03) VALUE 'NO '.

*****************************************************************
* Request/Response TSQ names.                                   *
* Primary/Secondary task ENQ List resource.                     *
*****************************************************************
 01  TSQ-LIST.
```

```
     02  TSQ-NAME           OCCURS 20 TIMES.
         10  TSQ-SYSID      PIC  X(04) VALUE SPACES.
         10  TSQ-TASKN      PIC S9(07) COMP-3.
         10  TSQ-REQ        PIC S9(04) COMP.
         10  FILLER         PIC  X(06) VALUE SPACES.
```

Each entry in the request array sent by the client consists of a *remote host ID* plus an *item ID*. The item ID is consistent across the array entries, but the remote host ID varies and is the variable by which distribution of the secondary tasks occurs. As the request queue entries are created by the primary task, an identifier consisting of the index position from the incoming request array plus the entry value (remote host ID plus item ID) is used in the registry queue as an association to the delegated secondary task that is started.

Example 6-5 provides an example for issuing the **WRITEQ TS** command for the request and registry queues and issuing the **START** command for the secondary task.

Example 6-5 Writing to the TSQ and starting secondary task

```
* This routine performs the following steps:                    *
* 1).  Write TSQ, for each Request for the Secondary task.      *
* 2).  Write TSQ, for each entry in the 'Registry'.             *
* 3).  START Secondary Subtask for each store request service.  *
*                                                               *
*****************************************************************
 0300-ATTACH-SUBTASK.
     MOVE SYSID                  TO TSQ-SYSID  (ATTACH-INDEX).
     MOVE EIBTASKN               TO TSQ-TASKN  (ATTACH-INDEX).
     MOVE ATTACH-INDEX           TO TSQ-REQ    (ATTACH-INDEX).

     MOVE TSQ-NAME(ATTACH-INDEX)  TO TSQ-QNAME.
     MOVE STORE-REQ(ATTACH-INDEX) TO TSQ-REQUEST(ATTACH-INDEX).

     EXEC CICS WRITEQ TS
          QNAME (TSQ-QNAME)
          FROM  (TSQ-REQUEST(ATTACH-INDEX))
          LENGTH(TSQ-LENGTH)
          NOHANDLE
     END-EXEC.

*****************************************************************
* Create a REGISTRY entry for each Secondary Subtask to provide *
* synchronization between Primary (web service provider) and    *
* Secondary (web service requester) tasks.                      *
*                                                               *
* The registry entry is set initially to 'NOTIFIED' and will be *
* set to 'REGISTERED' when the Secondary task receives control. *
*****************************************************************

     MOVE SYSID          TO TSR-SYSID.
     MOVE EIBTASKN       TO TSR-TASKN.
     MOVE 'NOTIFIED  ' TO TSQ-REGISTRY-DATA.
     MOVE 10             TO TSR-LENGTH.

     EXEC CICS WRITEQ TS
          QNAME (TSQ-REGISTRY)
          FROM  (TSQ-REGISTRY-DATA)
          LENGTH(TSR-LENGTH)
          NOHANDLE
     END-EXEC.

*****************************************************************
* Attach Subtask to initiate a web service request to a Store.  *
* Each Subtask will have its own unique TS queue.  The TSQ-REQ  *
* field is the unique request number of the TS queue for the    *
* task.  It is also the 'ITEM' number of the REGISTRY queue.    *
* Synchronization is achieved by using an 'ENQ' list similar    *
* in nature to an ECB list.                                     *
*****************************************************************

     EXEC CICS START
          INTERVAL(0)
          TRANSID (ATTACH-TRANSID)
          FROM    (TSQ-NAME(ATTACH-INDEX))
          LENGTH  (LENGTH OF TSQ-NAME)
     END-EXEC.

 0300-EXIT.
     EXIT.
```

Example 6-6 shows the secondary task that gets invoked, which updates the registry queue with a status that it is started.

Example 6-6 Sign registry

```
*****************************************************************
* Sign REGISTRY to notify Primary task that we've started.      *
*****************************************************************
 0150-SIGN-REGISTRY.
     MOVE SYSID     TO TSR-SYSID.
     MOVE TSQ-TASKN TO TSR-TASKN.

     EXEC CICS WRITEQ TS
          QNAME(TSQ-REGISTRY)
          FROM (TSQ-REGISTRY-DATA)
          ITEM (TSQ-REQ)
          REWRITE
          NOHANDLE
     END-EXEC.

 0150-EXIT.
     EXIT.
```

After the secondary tasks are dispatched, the primary task polls the registry queue to determine the status of the secondary tasks. This process is accomplished by checking the queue, and then linking to a custom program that performs a calculated delay before returning to the primary task. In this case, the program is named L8WAIT and issues the z/OS **STIMERM** command for 2.5 milliseconds. This operation is performed 40 times (for a total 100 milliseconds) before discarding the particular secondary task as a timeout (as shown in Example 6-7 on page 55).

Example 6-7 Check registry queue for each submask

```
*****************************************************************
* Check REGISTRY queue for each Subtask.  If all tasks have not *
* REGISTERED, wait 2.5 milliseconds a maximum of 40 REGISTRY    *
* scans for a total of 100 milliseconds.                        *
*****************************************************************
 0400-CHECK-REGISTRY.

     EXEC CICS LINK PROGRAM(L8WAIT) NOHANDLE
     END-EXEC.

     MOVE 'YES' TO ALL-TASKS-REGISTERED.

     PERFORM
         WITH TEST AFTER
         VARYING TSQ-INDEX FROM 1 BY 1
         UNTIL   TSQ-INDEX = StoreRequest-num

         MOVE LENGTH OF TSQ-REGISTRY-DATA TO TSQ-LENGTH

         EXEC CICS READQ TS
              QNAME (TSQ-REGISTRY)
              INTO  (TSQ-REGISTRY-DATA)
              LENGTH(TSQ-LENGTH)
              ITEM  (TSQ-INDEX)
         END-EXEC

         IF  TSQ-REGISTRY-DATA = 'NOTIFIED  '
             MOVE 'NO ' TO ALL-TASKS-REGISTERED
         END-IF

     END-PERFORM.

 0400-EXIT.
     EXIT.
```

Ultimately, the secondary task acquires the response from the remote host and writes this information to the response queue, where the primary task can then gather the response and deliver it back to the client. The response queue is updated via code, as shown in Example 6-8.

Example 6-8 Write response data

```
*****************************************************************
* Write Response data to TS queue for the Primary task.         *
* Move this information from the DFHWS-DATA container.          *
*****************************************************************
 0700-TSQ-RESPONSE.
     MOVE 1 TO TSQ-ITEM.

     EXEC CICS WRITEQ TS
          QNAME(TSQ-NAME)
          FROM (TSQ-RESPONSE)
          ITEM (TSQ-ITEM)
          NOHANDLE
     END-EXEC.

 0700-EXIT.
      EXIT.
```

By ensuring that the secondary task delegation is sysplex enabled and by using sysplex distribution of incoming requests, this type of workload can be spread across a sysplex for high-volume client requests to be processed simultaneously, as illustrated in Figure 6-6.

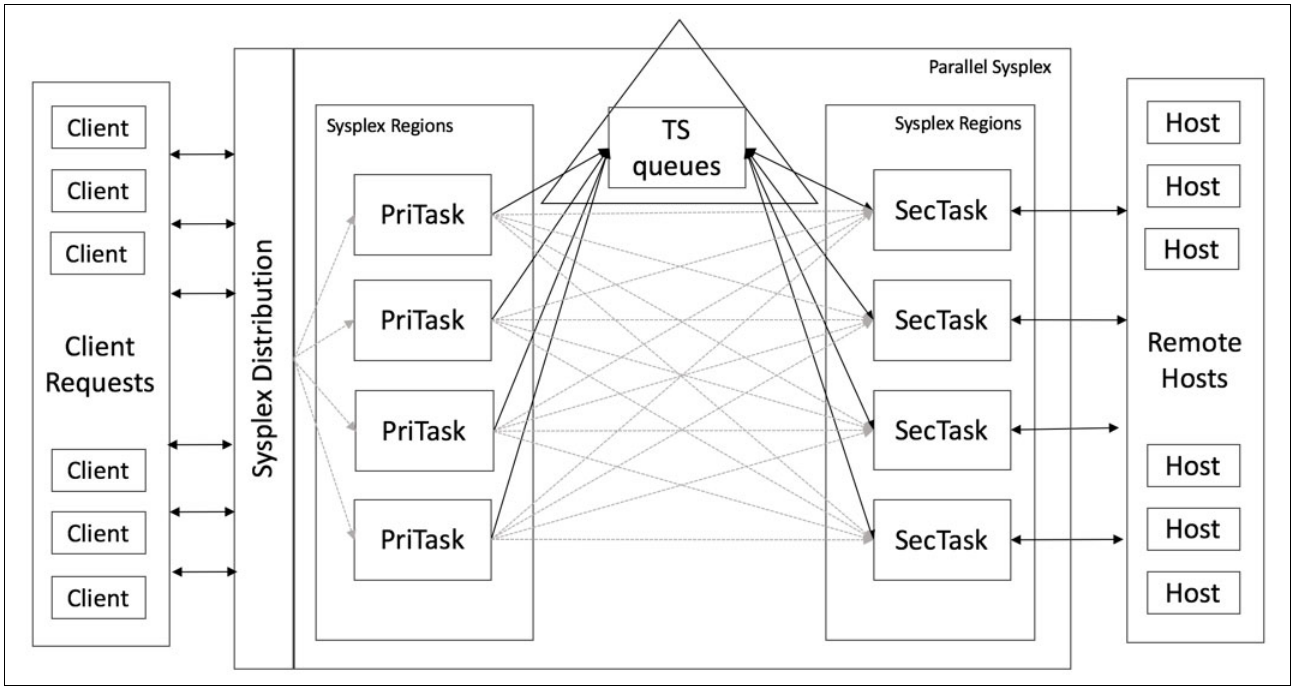


Figure 6-6 Sharing data among tasks to use a sysplex

The examples in this section demonstrate the core components for using shared TS to share data among CICS tasks for using a delegation of tasks across a Parallel Sysplex. Chapter 7, “Combined functional use” on page 57 includes details for supplementing these ideas with other capabilities, such as using GRS for serialization and CFDT for caching. The combined use of these capabilities provides a more complete and more robust solution to this type of problem.

7

Combined functional use

Each of the IBM Coupling Facility (CF) features discussed previously in this book can be used in interesting, and sometimes unconventional, ways. The use of these features can also be combined in various ways to provide solutions for any number of issues that you might need to address.

This chapter covers some examples of combining CF features to tackle a few assorted challenges. It includes the following topics:

- Affinities
- Sequencing captured updates for data synchronization solution
- Processing one-to-many parallel requests
- Tiered data store options
- Assist parallelization of downstream processing
- Using other CF resources
- Summary

7.1 Affinities

The main theme of this book is to show new, interesting, and useful ways of using CF resources to solve real-world problems. It also demonstrates that resources that reside in the CF are accessible by applications running anywhere in the sysplex rather than in a single or select group of CICS regions. The ubiquitous access to CF resources addresses a major issue faced by application developers when they look to make applications more highly available, namely that of *affinities*.

A CICS affinity is a situation that arises when CICS tasks share data in a manner that forces future tasks to return to a specific CICS or set of CICS regions. Specific examples of affinities include:

- An instance of a pseudo-conversational transaction that places some runtime data in a local temporary storage (TS) queue for use by a future instance of the transaction and creates an affinity to the CICS region that hosts the TS queue.
- A task uses a non-record-level sharing (RLS) Virtual Storage Access Method (VSAM) file to read customer account records can run only in a CICS regions that has access to that file.

Although application designers can implement preferred practices to avoid affinities, a need to share data always exists, and the ability to share that data throughout the sysplex through the CF removes many affinities with little or no changes to application code.

So, when reflecting upon the real-world examples in this book, it’s worth remembering that shared TS, named counters, a coupling facility data table (CFDT), and RLS are all good methods for removing application affinities.

“Affinity avoidance is a key requirement of highly available applications. Moving CICS resources into the CF plays a major part in removing such affinities.”
— Arndt Eade, IBM CICS Development Services Team

7.2 Sequencing captured updates for data synchronization solution

For this scenario, the issue involved synchronizing data updates between two application systems. As updates are made to one system, the same updates needed to be applied dynamically to the other system in real time, and vice versa. The application that this example is based upon deals with processing purchase orders. Thus, references in the examples to items such as “PO numbers” might display in the code samples. However, the design can be applied to any number of other issues. So most of the content is generalized as much as possible to maintain room for cognitive flexibility.

Some system exits were employed to capture the updates (through CICS and batch) and send the updates to a queue. Another CICS process appends some qualifying information to the updates and feeds them to a number of IBM Integration Bus processes for asynchronous transformation to XML. The transformed updates then need to be applied to another database but are no longer in sequence. However, a chronological replay of the events is necessary to maintain integrity of the target database.

The combination of CF technologies that were used to address this problem were VSAM RLS, CFDT, and global resource serialization (GRS), as illustrated in Figure 7-1 on page 59. Numerous CICS regions handle the processing and use several CFDTs.

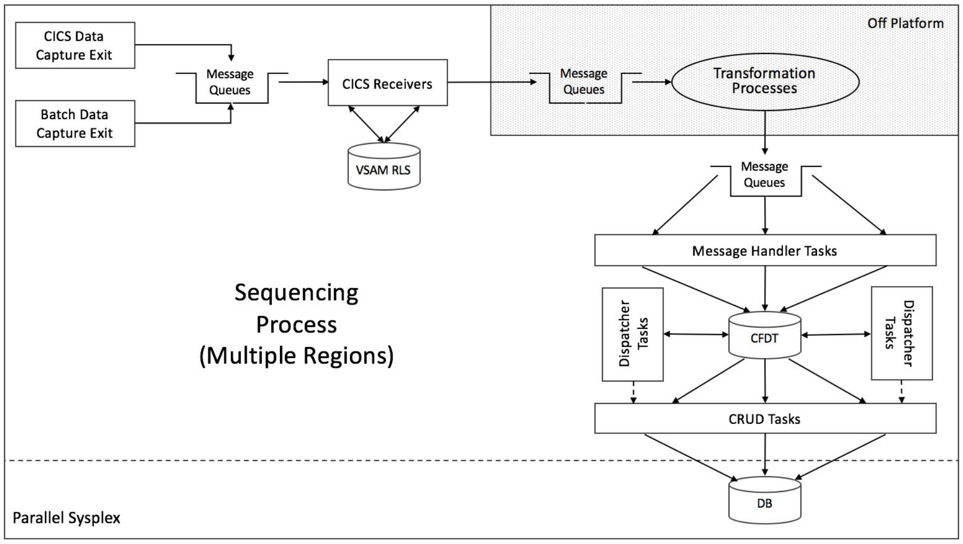


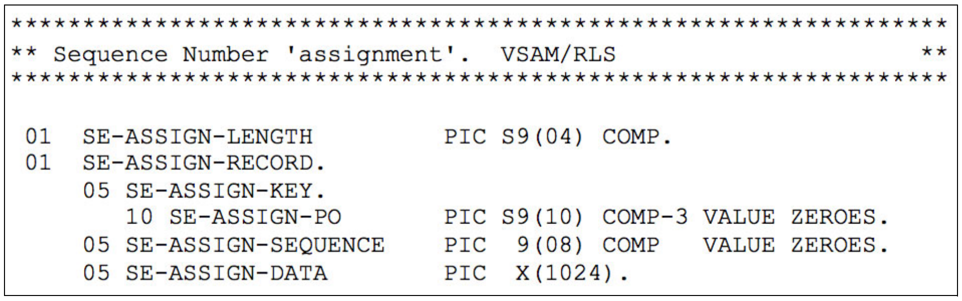
Figure 7-1 High-level view of sequencing solution

7.2.1 CFDT and VSAM RLS data stores

This solution uses several data stores, including VSAM RLS and CFDT. The sections that follow describe these data stores.

Sequence number assignment (VSAM RLS)

This table contains a sequence number associated with a particular update to a record. The data capture exits place the records onto a message queue. A receiver process in CICS takes the record from the queue and then accesses this VSAM RLS file to acquire and increment the sequence number related to the updated record. It then passes the information (with a composite key of the record number and the sequence number) to the transformation process via another message queue. Figure 7-2 shows an example of the record layout for this entry.



```
*****************************************************************
** Sequence Number 'assignment'.  VSAM/RLS                     **
*****************************************************************

 01  SE-ASSIGN-LENGTH         PIC S9(04) COMP.
 01  SE-ASSIGN-RECORD.
     05 SE-ASSIGN-KEY.
        10 SE-ASSIGN-PO       PIC S9(10) COMP-3 VALUE ZEROES.
     05 SE-ASSIGN-SEQUENCE    PIC  9(08) COMP   VALUE ZEROES.
     05 SE-ASSIGN-DATA        PIC  X(1024).
```

Figure 7-2 Sequence number assignment

Cache for records (CFDT)

This table is used to store and pass each record, which has a key that includes the record number and a sequence number and a payload that includes the required update information. A message handler process acquires these entries from the transformation process via message queues and then updates this CFDT. A dispatcher process references this table to identify entries that need to be processed and then attaches a create, retrieve, update, and delete task to perform the work. The create, retrieve, update, and delete task then grabs the entry from this CFDT and applies the necessary update to the target database. Figure 7-3 shows the record layout for this entry.

```
****************************************************************
** Sequence Engine 'cache' CFDT.                              **
****************************************************************

 01  PO-RECORD-LENGTH          PIC S9(04) COMP.
 01  PO-RECORD.
     05 PO-KEY.
        10  PO-CACHE-NUMBER    PIC S9(10) COMP-3 VALUE ZEROES.
        10  PO-CACHE-SEQUENCE  PIC  9(08) COMP   VALUE ZEROES.
        10  PO-CACHE-SEGMENT   PIC  9(02) COMP   VALUE ZEROES.
        10  PO-CACHE-SUFFIX    PIC  9(08) COMP   VALUE ZEROES.
     05  PO-TOTAL-SEGMENTS     PIC  9(02) COMP   VALUE ZEROES.
     05  PO-CACHE-ABS-TIME     PIC S9(15) COMP-3 VALUE ZEROES.
     05  PO-CACHE-DATA         PIC  X(32000).
```

Figure 7-3 Sequence engine cache

Next sequence number (CFDT)

This table is referenced to maintain chronology during processing. It contains entries that identify the next sequence number to process for a given record number. The dispatcher process checks this table to determine whether the record in cache matches the next sequence number to be processed. If the sequence number is a match, the dispatcher process attaches a create, retrieve, update, and delete task to perform the work. The create, retrieve, update, and delete task then increments the next sequence number in this CFDT.

If the sequence number is not a match, the task stops and the dispatcher process ultimately attaches a new create, retrieve, update, and delete task when there is a match. If an entry number simply doesn't exist in this table for a respective cache entry, the dispatcher process creates an entry with sequence number 1 for this record and proceeds with the create, retrieve, update, and delete attachment. Figure 7-4 shows the record layout for this entry.

```
******************************************************************
** Sequence Number 'next' CFDT.                                 **
******************************************************************

 01  SE-NEXT-LENGTH            PIC S9(04) COMP.
 01  SE-NEXT-RECORD.
     05 SE-NEXT-KEY.
        10 SE-NEXT-PO          PIC S9(10) COMP-3 VALUE ZEROES.
     05 SE-NEXT-SEQUENCE       PIC  9(08) COMP   VALUE ZEROES.
```

Figure 7-4 Sequence number next

Figure 7-5 illustrates a generalized view of this overall process, and the data stores involved.

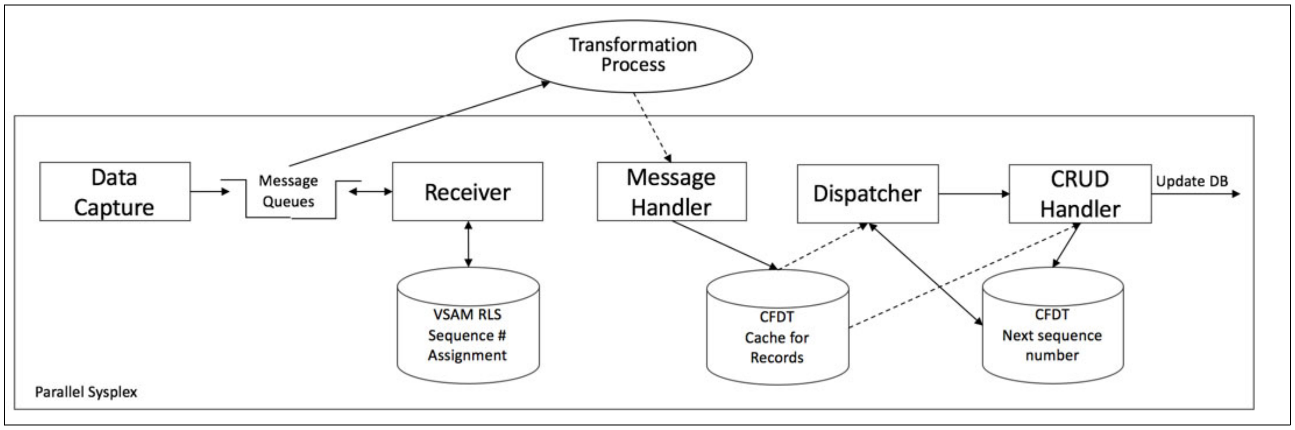


Figure 7-5   Sequencing process and associated data stores

Operational control (VSAM RLS and CFDT)

Aside from the core record-passing components of this solution, another data store serves an important role in this design. This table provides a centralized location for operational configuration settings for the solution. The processes in each region consult this table continually and adjust their actions to accommodate the settings in the configuration. This process allows real-time administration for the solution.

An administrative transaction is used to control all of the processes that are involved in this sequencing and synchronization solution. This administrative transaction has approximately 75 control parameters that are used to granularly manage how the process executes. These configuration parameters are stored in a VSAM RLS data store to ensure that a system of record is always persisted to disk. However, they are also loaded into a CFDT for the ongoing real-time accessibility to ensure the most performant and current access for the various components of the solution.

A few examples of the control parameters include:

- Sleep parameters

  The application runs continually, always capturing and synchronizing updates. However, there are occasions, such as during maintenance activities, where temporarily stopping the processing can be useful. These parameters provide a time and duration based invocation for stopping parts of or the entire application.

- MQ throttle

  The application includes functionality for stopping and starting MQ channels to manage the throughput of the message handling. These actions are determined by configuration parameters that are associated with a queue depth. When a specified *high-value* percentage of the maximum queue depth is exceeded, channels are stopped to restrict the rate of messages flowing through the system. Conversely, when the quantity of messages falls below a specified *low-value* percentage of the maximum queue depth, additional channels are started to increase the processing throughput.

- Active tasks

  The application also provides the ability to dynamically manage the number of active tasks in a region or across the entire sysplex as described in 4.3.1, “Global transaction class” on page 23.

7.2.2 Global resource serialization

GRS is used within this solution for a couple of purposes. The primary use of GRS is traditional serialization purposes. It is used to both serialize the assignment of individual record processing to a particular create, retrieve, update, and delete task and to ensure the processing of each record by the create, retrieve, update, and delete task is properly serialized to maintain integrity of the sequenced updates to the database. Details and code samples for this type of usage are covered in 4.3.2, “Serialization for dispatched tasks” on page 25 and should be reviewed.

Figure 7-6 provides an updated view that reflects the current context more directly.

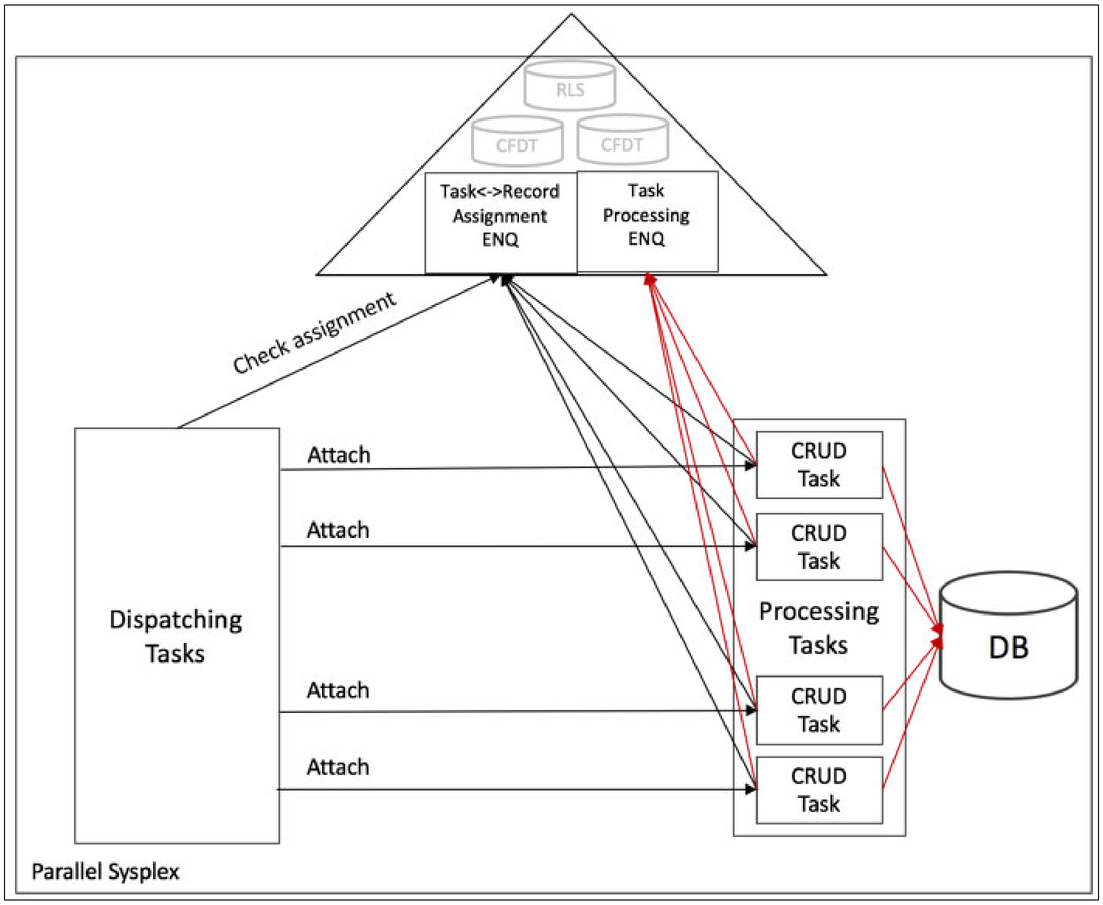


Figure 7-6 Using GRS for serialization activities

GRS is also used to provide a control on the total number of concurrent transactions to be running in the Parallel Sysplex. This concept is similar in nature to the region-level TRANCLASS attribute but applies to transactions that are running in numerous regions across the sysplex. The mechanics of this process and some related code samples are covered in 4.3.1, “Global transaction class” on page 23.

Within the context of the application being currently discussed, the maximum concurrent task value is acquired from the Operational Control CFDT, and the dispatcher uses the Global TranClass capability when attaching the create, retrieve, update, and delete tasks.

Figure 7-7 provides a supplementary view of the design within this setting.

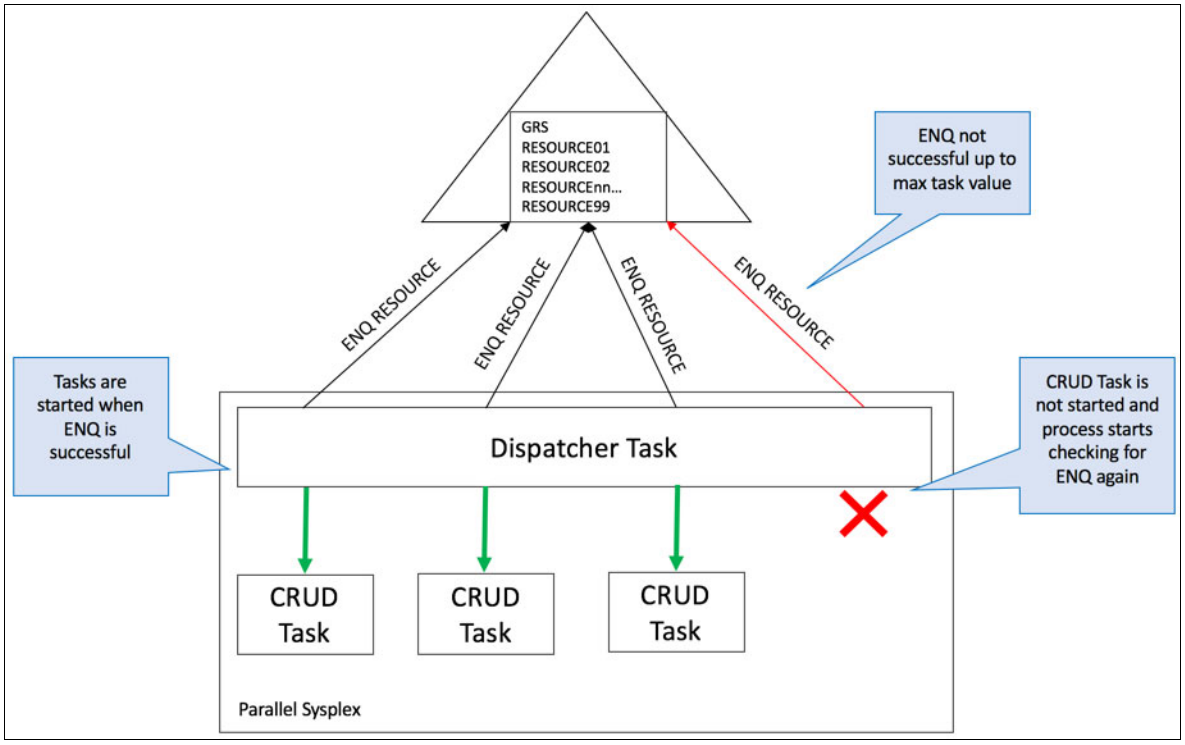


Figure 7-7   Using GRS for Sysplex-wide transaction class

7.3 Processing one-to-many parallel requests

In this scenario, a single incoming client request triggers numerous other parallel requests that, in turn, become clients to remote sources to gather information. That information from the numerous remote sources then needs to be packaged together and returned to the originating requestor client.

Figure 7-8 shows a highly simplified view of this process.

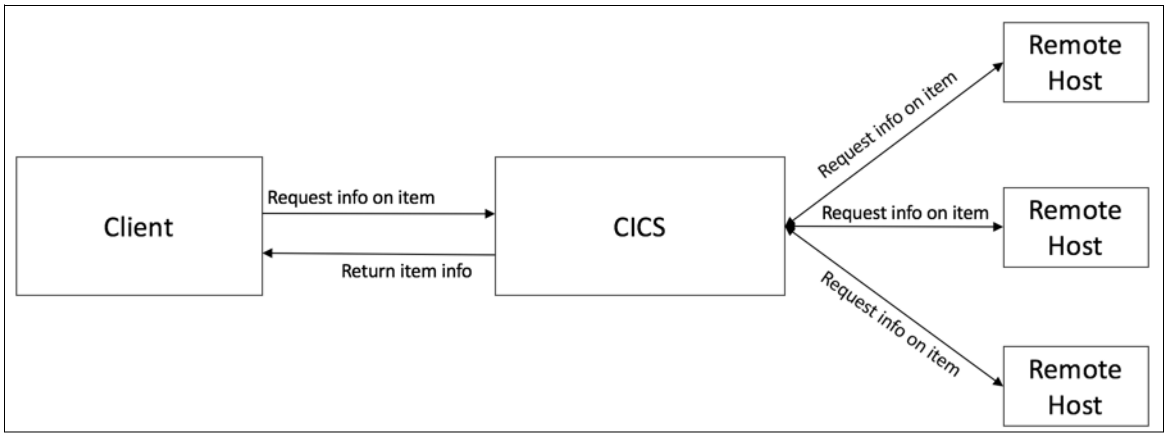


Figure 7-8   High-level view of parallel processing

In the application that this use case is based upon, all of the requests, including the originating client request and the subsequent requests to the remote hosts, are web service calls. The incoming request is taken by a single primary CICS task, and then secondary tasks are initiated to perform each of the parallel requests that call a service on a remote host. The responses from those parallel service requests are provided back to the original primary task, which in turn provides the aggregated responses to the requestor.

For the implementation of this pattern at Walmart, the initiating client request primarily comes from web-based endpoints (such as a website or a mobile app), and the request is for information about particular items from the remote hosts, which are systems located within the brick and mortar stores. Particularly, a client requests information about an item that is identified by a universal product code (UPC) and information about that UPC, such as price or inventory levels, is requested from a number of remote hosts and returned to the client. Although this implementation is particular to a retail business setting, this pattern can be employed for any scenario that exists with a similar topology.

This solution uses the following combination of CF-related technologies:

- Shared TS for various forms of passing information between tasks
- GRS for serializing activities against some of the shared TS queues
- CFDT for caching certain data

Figure 7-9 illustrates the overall design a bit more completely.

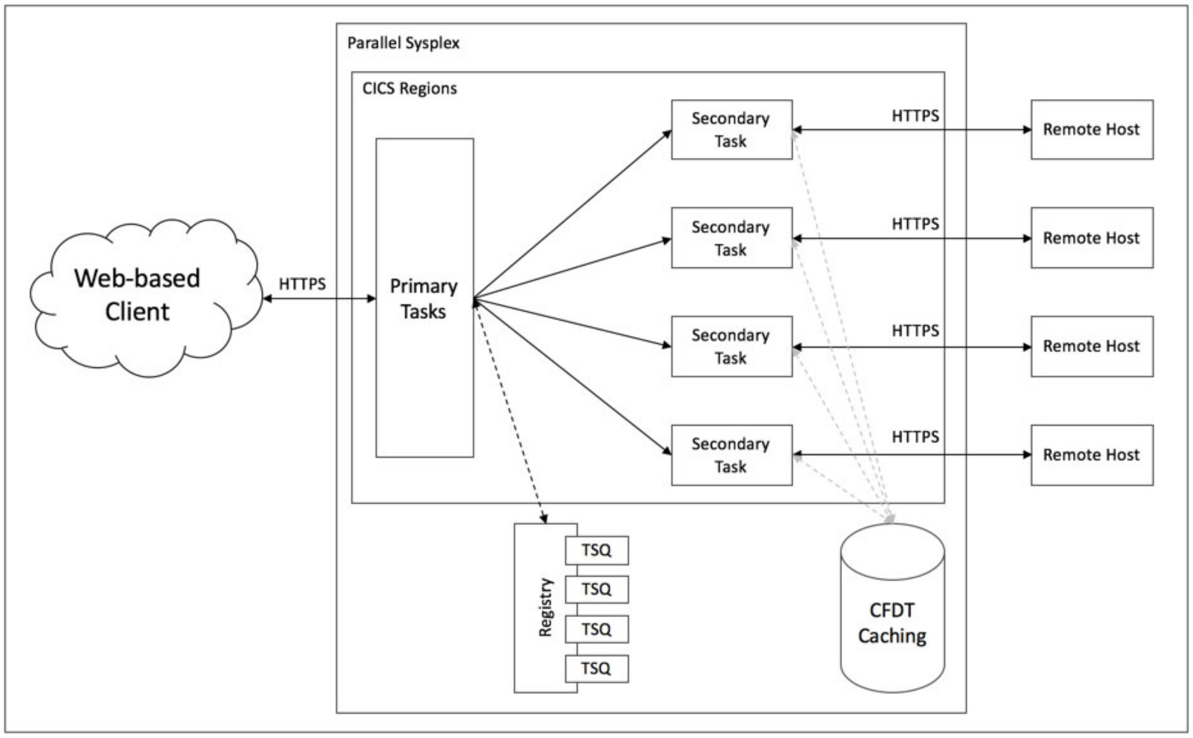


Figure 7-9 Solution design of parallel processing

The following sections delve into the roles that each CF technology plays in this overall solution.

7.3.1 Shared TS queues

Shared TS queues are integral to this design. At a high level, this problem consists of one synchronous process (the initiating client request) and other synchronous processes (requests to remote hosts) that are separated by an asynchronous process (that delegate the multiple parallel remote host requests). Managing these disparate requests is accomplished by a correlating process that uses a few shared TS queues.

The following shared TS queues are used for this purpose:

- The Registry TS queue

  This queue provides a location to keep track of the status of each delegated task. The primary task logs a delegated task to this queue. The secondary task updates this entry with completion status. The primary task also polls this queue to determine status of the delegated task.

- The Request TS queue

  The information related to the individual request to be processed by a secondary task is placed on this queue by the primary task. The secondary task grabs this entry to execute the request to the remote host.

- The Response TS queue

  The secondary task places the response from the remote host onto this queue. The primary task (when a status update on the Registry queue is made) retrieves the response from this queue then delivers it to the client.

Figure 7-10 illustrates the basic flow of an individual request delegation.

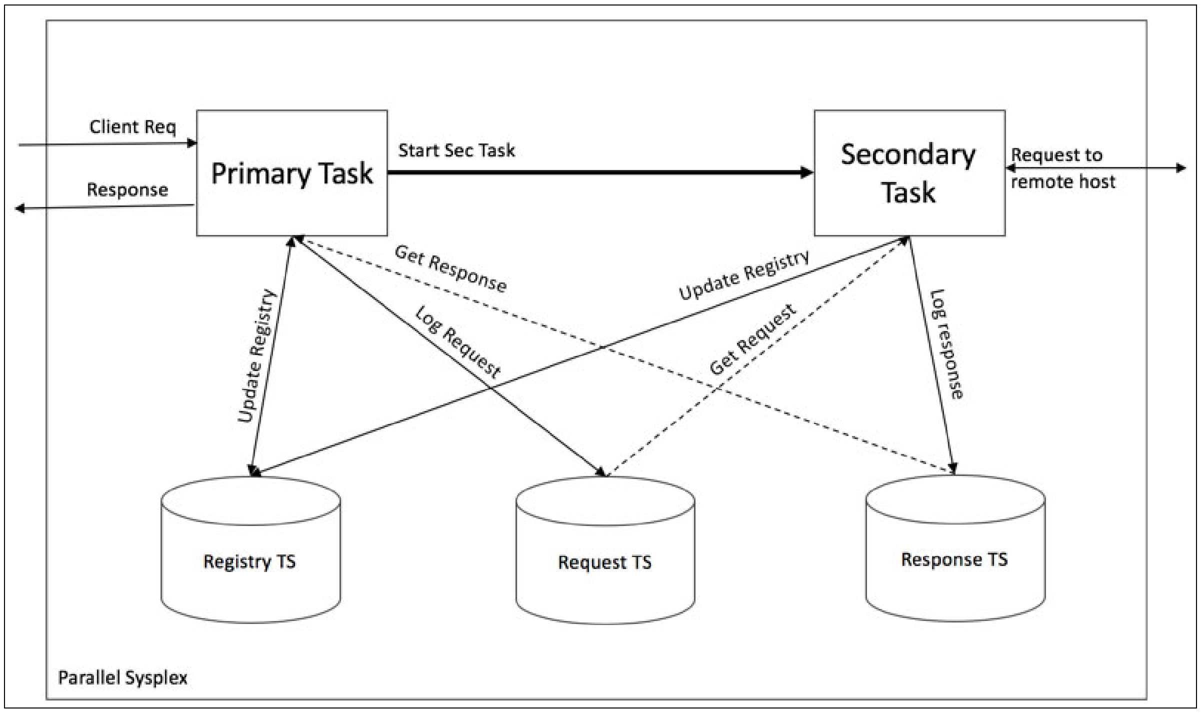


Figure 7-10 Using shared TS queues to associate disparate synchronous requests

This correlation process allows synchronization of the disparate synchronous processes that are associated with the request. By using composite identifiers of request information and remote host IDs for the TS queue entries, the correlating process provides the ability to execute numerous secondary tasks in parallel that can be processed by a single primary task. And by using shared TS queues, the secondary task delegation is not confined to the region that is running the primary task, so the work can be distributed to regions throughout the sysplex, as illustrated in Figure 7-11.

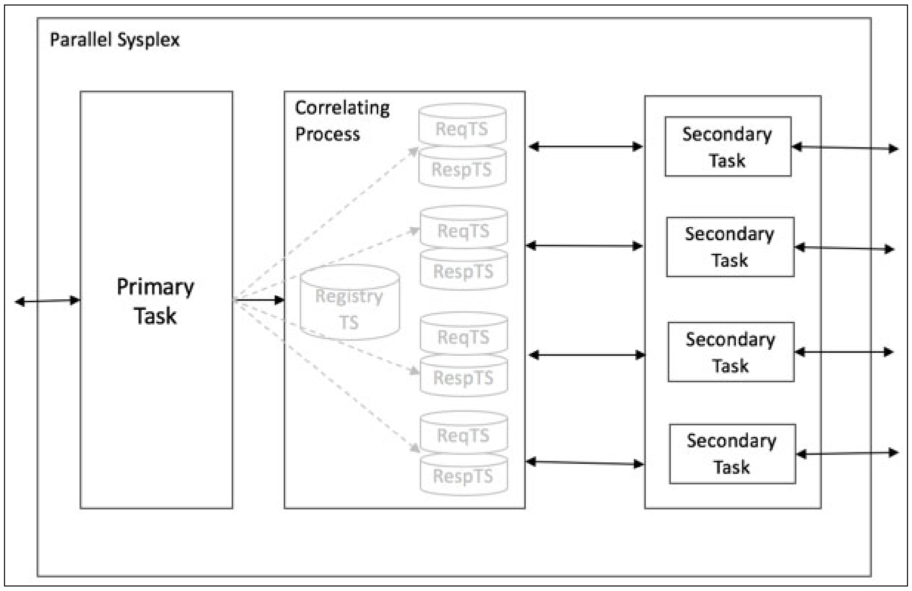


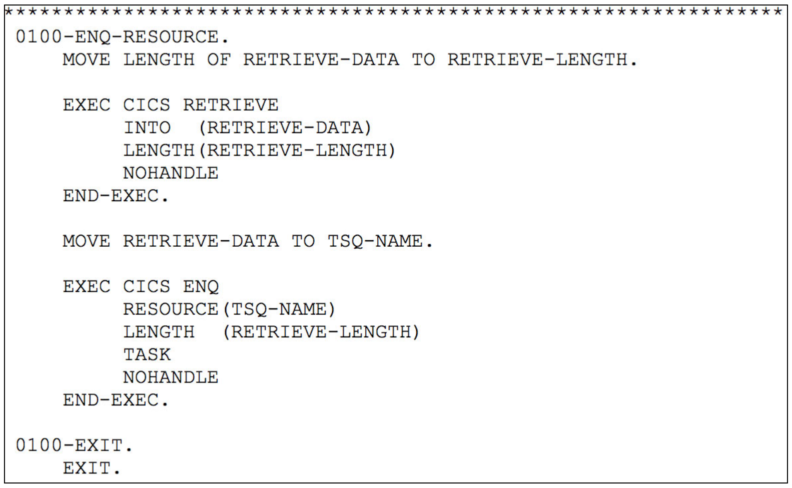
Figure 7-11 Using shared TS queues to deploy parallel secondary processes

You can find additional details and code samples related to this process in 6.3.2, “Sharing data among tasks” on page 51. For an alternative method to manage CICS asynchronous processes, refer to *IBM CICS Asynchronous API: Concurrent Processing Made Simple*, SG24-8411.

7.3.2 GRS

Although the shared TS queues provide the underlying mechanism for tying the client request and the remote host requests together, additional coordination of control is needed to ensure the integrity of the overall process. This coordination is accomplished by using GRS in a creative way.

When each secondary task starts, it issues an **ENQ-RESOURCE** against the shared TS queue that it is processing, as demonstrated in Figure 7-12.

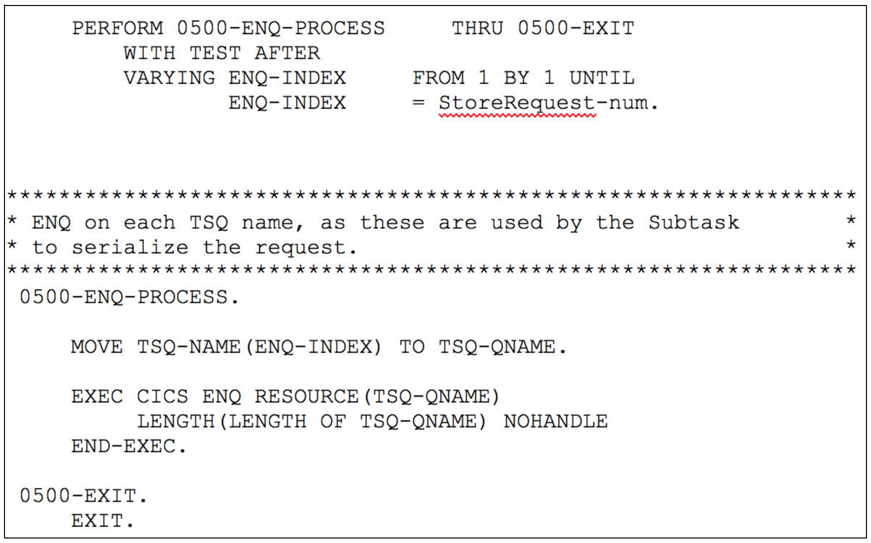


```
*****************************************************************
 0100-ENQ-RESOURCE.
     MOVE LENGTH OF RETRIEVE-DATA TO RETRIEVE-LENGTH.

     EXEC CICS RETRIEVE
          INTO   (RETRIEVE-DATA)
          LENGTH(RETRIEVE-LENGTH)
          NOHANDLE
     END-EXEC.

     MOVE RETRIEVE-DATA TO TSQ-NAME.

     EXEC CICS ENQ
          RESOURCE(TSQ-NAME)
          LENGTH   (RETRIEVE-LENGTH)
          TASK
          NOHANDLE
     END-EXEC.

 0100-EXIT.
     EXIT.
```

Figure 7-12   Issuing ENQ-RESOURCE

The secondary task next updates the Registry TS that it has started processing this entry. Meanwhile, the primary task checks the Registry TS queue to ensure that all respective secondary tasks have updated their status to indicate that they've started and then starts through an ENQ process on each Request TS queue that is associated with the request array to see whether the secondary task has completed the processing of the individual task.

Figure 7-13 shows sample code that is associated with this activity.



```
     PERFORM 0500-ENQ-PROCESS      THRU 0500-EXIT
         WITH TEST AFTER
         VARYING ENQ-INDEX      FROM 1 BY 1 UNTIL
                 ENQ-INDEX      = StoreRequest-num.



*****************************************************************
* ENQ on each TSQ name, as these are used by the Subtask        *
* to serialize the request.                                     *
*****************************************************************
 0500-ENQ-PROCESS.

     MOVE TSQ-NAME(ENQ-INDEX) TO TSQ-QNAME.

     EXEC CICS ENQ RESOURCE(TSQ-QNAME)
          LENGTH(LENGTH OF TSQ-QNAME) NOHANDLE
     END-EXEC.

 0500-EXIT.
     EXIT.
```

Figure 7-13   Perform the ENQ-PROCESS

As the secondary tasks release their enqueues on the Request TS queues, the primary task can acknowledge the completion and include the response with the payload. The result of this process is similar to checking an ECBLIST but is applicable at the sysplex scope instead of a singular region scope. Additional details related to this approach are available in 4.3.3, "Enqueue list" on page 28.

7.3.3 CFDT

The CFDT is used as a cache in this application. Basically, each secondary task that is invoked first checks the CFDT cache for a result to the specific request that it has received. If it finds the information in the CFDT cache, it immediately returns it to the primary task; otherwise, it makes the call to the remote host, logs the returned response to the CFDT cache, and then provides the response to the primary task.

All of the previously discussed actions related to message passing and serialization still occur during this process. The secondary task performs the cache check just before issuing the remote host call and, in the event of a cache-miss, updates the CFDT before updating status or releasing the associated ENQ.

For reference, Figure 7-14 provides an example of checking the cache.

```
*****************************************************************
 0300-CHECK-CACHE.
     MOVE TSQ-REQ-STORE       TO CF-STORE.
     MOVE TSQ-REQ-UPC         TO CF-UPC.

     EXEC CICS READ
          FILE  (CF-FILE)
          INTO  (CF-RECORD)
          RIDFLD(CF-KEY)
          LENGTH(CF-LENGTH)
          RESP  (CF-RESPONSE)
          NOHANDLE
     END-EXEC.

     IF  CF-RESPONSE EQUAL DFHRESP(NORMAL)
         MOVE CF-STORE        TO TSQ-RESP-STORE
         MOVE CF-UPC          TO TSQ-RESP-UPC
         MOVE CF-PRICE        TO TSQ-RESP-PRICE
         MOVE CF-ONHAND       TO TSQ-RESP-ONHAND
         MOVE CF-DESC         TO TSQ-RESP-DESC.

 0300-EXIT.
      EXIT.
```

Figure 7-14 Checking the cache

Figure 7-15 provides an example of updating the CFDT cache.

```
******************************************************************
 0650-WRITE-CACHE.
     MOVE TSQ-RESP-STORE         TO CF-STORE.
     MOVE TSQ-RESP-UPC           TO CF-UPC.
     MOVE TSQ-RESP-PRICE         TO CF-PRICE.
     MOVE TSQ-RESP-ONHAND        TO CF-ONHAND.
     MOVE TSQ-RESP-DESC          TO CF-DESC.

     EXEC CICS WRITE
          FILE  (CF-FILE)
          FROM  (CF-RECORD)
          RIDFLD(CF-KEY)
          LENGTH(CF-LENGTH)
          RESP  (CF-RESPONSE)
          NOHANDLE
     END-EXEC.

 0650-EXIT.
     EXIT.
```

Figure 7-15 Updating the cache

You can find additional details about using CFDT for caching purposes in 5.3.1, “Caching information” on page 34.

7.4 Tiered data store options

You can find information about the flexibility that is provided by the CFDT and the consistency of the CICS file access APIs in 5.3.3, “Flexibility for data stores” on page 41. That section describes a scenario where an upfront decision determines the location of a data store. However, it is possible to use CFDT in conjunction with RLS to adjust data stores dynamically in some situations.

The application of focus here is the object store service (zFAM) that is described in 3.3.2, “Generating components of keys” on page 15. The underlying file system structure of that service includes a RLS key-sequenced data set (KSDS) that stores the user keys and one or more other RLS KSDSs to store the pieces of the objects (up to 2 GB in size) that have been split into 32 KB segments. The KSDS that houses the user keys includes internal keys that point to the associated segments of the object in the other KSDSs.

Figure 7-16 shows a simple view of this structure.

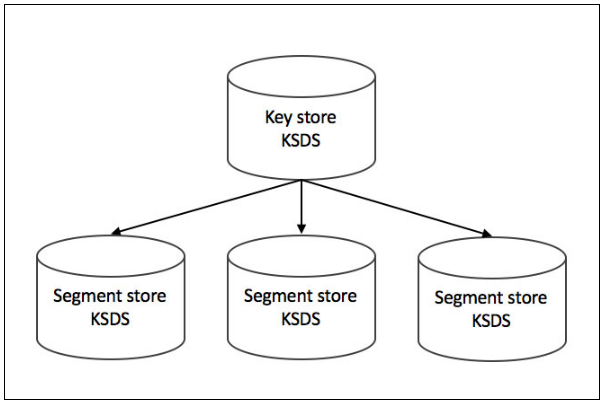


Figure 7-16 File system structure of object store service

Another component of the internal key structure includes a DD name that is associated with the individual segment store KSDSs. This is what provides the ability to use multiple files for this purpose. Additional segment stores can also be added dynamically to accommodate increasing space requirements. This capability enables us to also add segment stores of a different type, such as CFDT.

One installation of this service initially went live with a fairly typical read/write ratio. Over time, the usage pattern on this service instance became increasingly write heavy, which led to constraining space on the existing RLS structures and a concern for continued delivery of acceptable performance.

To address this issue, additional CFDT-based segment stores were added dynamically for the service instance to alleviate space constraints and to also provide a faster I/O path for the increasingly write-heavy usage pattern. This change is achieved by simply adding the new CFDT-based file definitions to the service instance (the code itself is written to use all defined segment files.

Figure 7-17 represents a simple view of the dynamically updated configuration.

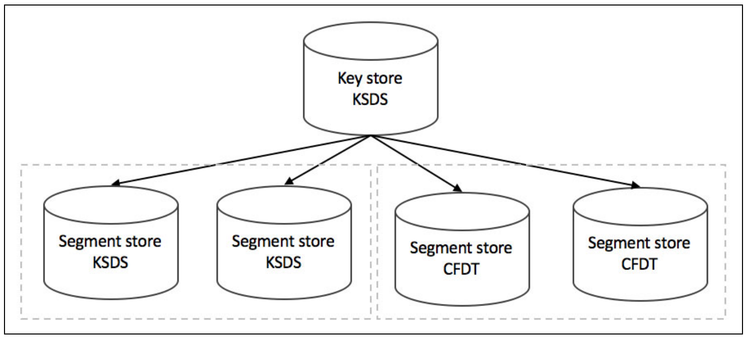


Figure 7-17 Differing data store structures for single service

The application discussed in this section is available in its entirety from GitHub, but the provided resource definition online (RDO) file definitions are all set up as KSDS. To include CFDT file definitions, adjust the RDO to include the `TABLE()` and `POOLNAME()` values as described in 5.2.3, "Defining CFDTs in CICS" on page 33. Although not required, you might consider also adjusting the DD name to reflect that it is related to a different type of data store. For example, using a DD name such as CFDSxx for CFDT files and VSDSxx for VSAM can facilitate identifying the type from just a list of the DD names.

7.5 Assist parallelization of downstream processing

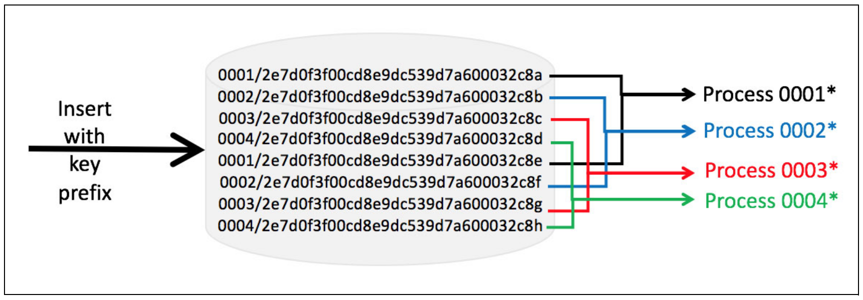
As discussed in Chapter 3, "Named counters" on page 11, the named counter can be useful in key generation, which is applicable in relation to objects being stored in an RLS data store. In many cases, the objects that are stored need to be processed in some way downstream. To facilitate parallelization of these downstream processes (for example, background transactions, batch jobs, and so forth), you can also use the named counter—along with previous named counter use—to assign a number of key prefixes that provide the ability to process arbitrary key groupings.

This example uses the zUID. It relies on a named counter to generate a key for new objects that are inserted into the RLS-based object store, which is referenced in several sections of this book. It also employs a feature that uses the named counter to prefix the unique keys with an integer, 1 through 4. It simply rotates through that sequence repeatedly for every inserted object, as listed in Table 7-1.

Table 7-1 Named Counter key prefix

Action	**Resulting key**
Insert object 1	`0001/2e7d0f3f00cd8e9dc539d7a600032c8a`
Insert object 2	`0002/2e7d0f3f00cd8e9dc539d7a600032c8b`
Insert object 3	`0003/2e7d0f3f00cd8e9dc539d7a600032c8c`
Insert object 4	`0004/2e7d0f3f00cd8e9dc539d7a600032c8d`
Insert object 5	`0001/2e7d0f3f00cd8e9dc539d7a600032c8e`
Insert object 6	`0002/2e7d0f3f00cd8e9dc539d7a600032c8f`
Insert object 7	`0003/2e7d0f3f00cd8e9dc539d7a600032c8g`
Insert object 8	`0004/2e7d0f3f00cd8e9dc539d7a600032c8h`

Then parallel downstream processes can ingest groups of keys, based on a key prefix and wildcard (for example, `0001*`, `0002*`, and so on.), without contention or logic for distributing the processing, as shown in Figure 7-18.



Insert with key prefix
0001/2e7d0f3f00cd8e9dc539d7a600032c8a
0002/2e7d0f3f00cd8e9dc539d7a600032c8b
0003/2e7d0f3f00cd8e9dc539d7a600032c8c
0004/2e7d0f3f00cd8e9dc539d7a600032c8d
0001/2e7d0f3f00cd8e9dc539d7a600032c8e
0002/2e7d0f3f00cd8e9dc539d7a600032c8f
0003/2e7d0f3f00cd8e9dc539d7a600032c8g
0004/2e7d0f3f00cd8e9dc539d7a600032c8h
Process 0001*
Process 0002*
Process 0003*
Process 0004*

Figure 7-18 Downstream processing of key groups based on prefix

The capability described in this section is available in the object store service zFAM that is referenced throughout this book. The complete code that is associated with this product is available for use or reference in GitHub, and you can find details about the particular features that are described in this section in the associated documentation. Review the sections about the zFAM-UID and zFAM-Modulo HTTP.

7.6 Using other CF resources

This book focuses on CF technologies that are directly usable by CICS. Other CF-based features are relevant to the solutions that are described here but are beyond the scope of this document in regards to a detailed discussion. However, it is worth mentioning those features briefly due to their relevance to the discussed solutions and in the spirit of encouraging further consideration for creative use of CF technologies.

7.6.1 Shared queues

The solutions described in 7.2, “Sequencing captured updates for data synchronization solution” on page 58 include references to *message queues* in general or to IBM MQ specifically. Note that these references assume an implementation that uses queue sharing groups to complement the availability and scalability characteristics expected from using the CF to begin with. However, the concept of the sysplex-wide shared queue is a compelling concept that might be taken advantage of in creative ways itself.

7.6.2 Workload Manager-managed sysplex distribution

Using Workload Manager (WLM)-managed sysplex distribution is touched upon in 2.3.2, “Improved scalability” on page 9. This design feature is integral to virtually all of the solutions described in this book. It can be extremely valuable approach to distributing incoming work through sysplex resources. Consider it seriously for any design decisions that are related to truly taking advantage of the Parallel Sysplex.

7.7 Summary

The material in this book includes numerous examples of ways to use various CF features with CICS and takes a look at how you can combine these features to develop interesting solutions to real-world problems. Although this content is intended to educate, it can also hopefully inspire. In what other unusual ways might these features be used? How might they be used in complementary fashions to build new capabilities? These are unique technologies, and with a little imagination, you can apply them to achieve special and valuable functionality.

SG24-8420-00

ISBN 0738443042

Printed in U.S.A.