Component Design Specification for the MIT Multi-Threaded X Window Sample Server

Mike Haynes
Paul Layne
Rick Potts
John A. Smith

Hiroya Chiba
Akio Harada
Hidenobu Kanaoka
Takayuki Miyake

Data General Corporation
62 T.W. Alexander Drive
Research Triangle Park, NC 27709

Omron Corporation

Kyoto, Japan

in association with
MIT X Consortium
545 Technology Square
Cambridge, MA 02139

Version 5.0
April 15, 1993

NOTICE

A vertical bar in the margin of a replacement page indicates
substansive technical change from the previous revision.

CHAPTER 1 Introduction

1.1 Revision History

1.0 Initial draft version, September 6. 1991

2.0 Second draft version, January 9, 1992

- Add slight changes to the MST.
- Moved validation functionality from the CCT to the CIT.
- Add new connection validation changes to the CIT.
- Add a ticket taking scheme for message delivery to/from the COT buffers. This will integrate more fully with event processing when it is defined in the third draft version of this document.
- Completely rewrote the resource locking strategy.
- Add chapter on DDX issues (CHAPTER 14).
- Add screensaver and block/wakeup handler strategies (CHAPTER 15).
- Note: event processing may get redesigned. This redesign will be incorporated in the third draft version.

3.0 Third draft version, June 6, 1992

- Simplify the layout of the chapters.

- Add details to MST, CCT, CIT.
- COT is created dynamically.
- Complete description of message delivery and ticket taking.
- Replace Window Lock List with Pending Operation Queue.
- Event processing does not need redesign; only re-implementation.
- Change details of DDX issues.
- Add Message Object Monitor chapter.
- Add Extension Writers Guidelines chapter.

4.0 Fourth draft version, October 23, 1992

- Replace ticket taking with local message buffering
- General threads chapter cleanup.
- Rewrite all locking chapters.
- Rewrite Extension Writers Guidelines chapter.

5.0 Fifth draft version, April 15, 1993

- Bring entire document up to date with evolving design.
- Add Signal Handling Thread chapter.
- Add Font Locking sections.

1.2 Purpose

This Component Design Specification (CDS) document provides a detailed description of the major components of the Multi-Threaded X (MTX) server and shows how the MTX server will be implemented. The CDS is the third in a series of documents that will describe the MTX server. These documents follow closely the software development cycle of functional analysis, functional design, detailed design, and implementation. (NOTE: in some development organizations, the CDS is also called the Detailed Design Specification (DDS).)

Although some readers may not be familiar with a software design methodology, such as Yourdon [MPJ88, Y84,Y85], we encourage you to become acquainted with these engineering tools so that you may better understand the MTX server design.

The MTX server will be provided by the MIT X Consortium as a sample implementation using POSIX 1003.4a threads and conforming to the X Protocol as specified in [SG90].

The design specified in the CDS is derived from the functionality specified in the MTX Functional Design Specification (FDS). Whereas the FDS said “what it does”, the CDS will be a basis for implementing the MTX server and tell implementors “how it is done”.

1.3 Major Chapter Sections

Each chapter that describes an MTX component is organized into the following sections. A component is defined to be either a thread or a monitor. The thread encapsulates code and dictates flow of control. The monitor encapsulates objects and dictates access control.

1. Overview
   This section presents a high-level description of the component. It explains the major functions of the component and how they generally relate to the other components of the MTX server.
2. Data Objects
   Describes the data objects manipulated by the component. These objects can include tables, queues, lists, global variables, etc. The interrelationship of these objects is also described. The use of any locks associated with the data objects is also described.
3. Concurrency Issues
   This section explains the major concurrency issues that affect the component. These issues include thread creation/deletion, mutex locking/unlocking, and interactions of this component with other components via synchronization primitives (i.e. wait/signal).
4. Internal Organization and Implementation
   Explains the overall organization of the component in a detail sufficient to start coding the implementation. This section describes how this component interfaces to the different functional levels (such as DIX, OS, DDX, and MI). This section also discusses the MTX server implementation and how it differs from X11R5. FIGURE 1 shows the notation to be used in describing these differences. Note that double barred boxes indicate system and library routines rather than MTX server routines.

.

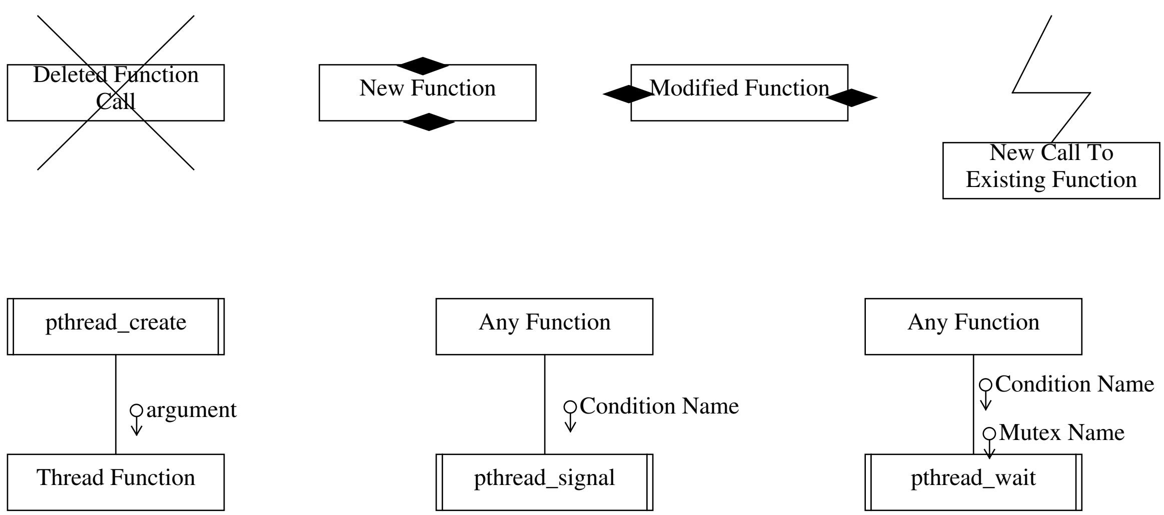

FIGURE 1 Structure Chart Notation

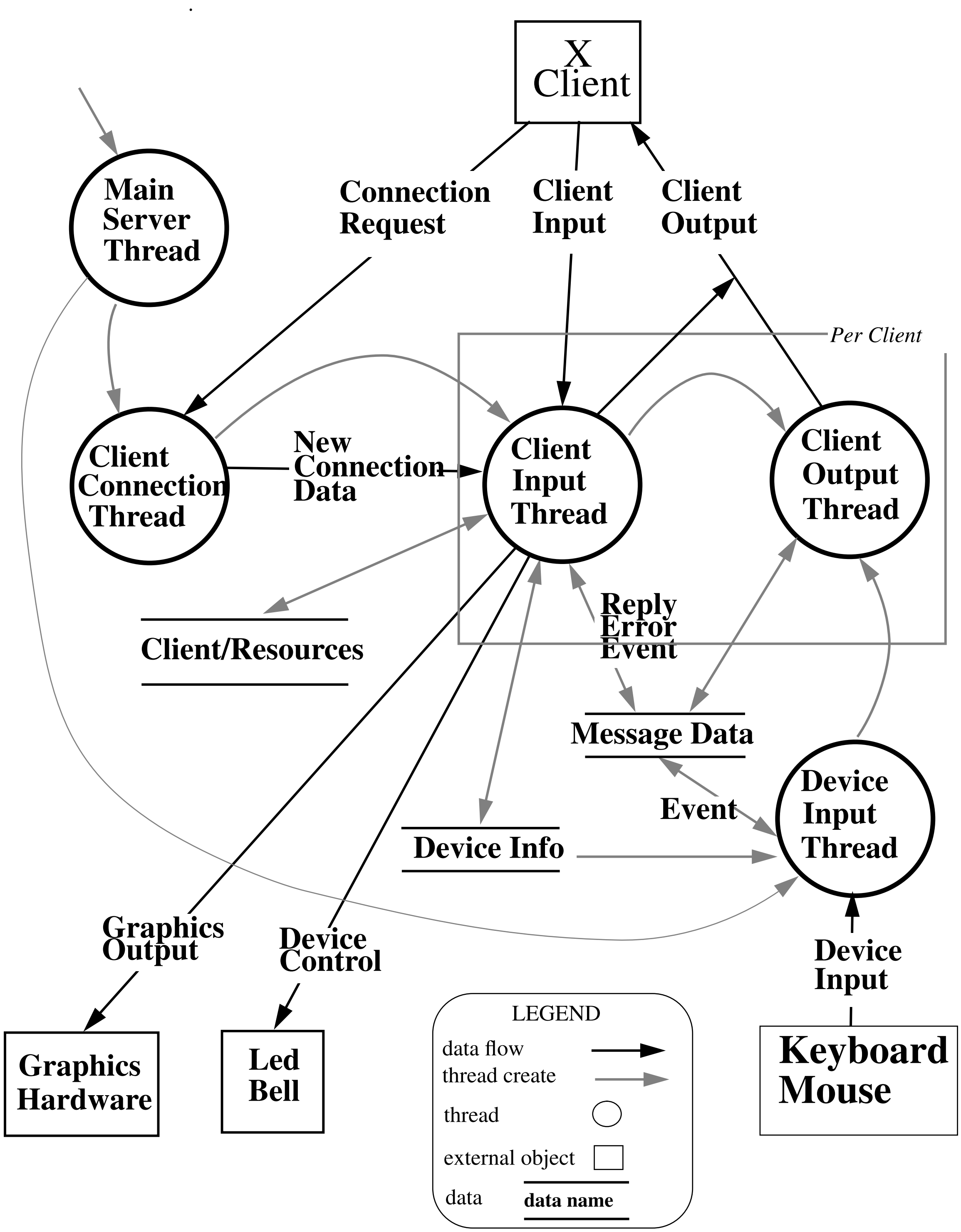

FIGURE 2 MTX Server Threads

1.4 MTX Overview

The functionality of the MTX server will be the same as that of the current R5 sample server. Although the functionality is equivalent, the implementation is not. The MTX server will be implemented with threads and concurrency support whereas the R5 server has a single thread of control.

The current X server looks for client requests, input events, and new client connections within the *dispatch* code. By combining these three concurrent functions into one serial loop, the interactivity of the R5 server suffers. Using threads, we can divide these functions into separate flows of control. The general strategy is to not bind functionality into threads such that there is a strong inter-thread coupling. Tightly coupled threads require more code and synchronization overhead to support inter-thread communication (ITC). This additional complexity makes debugging much more difficult and reduces modularity.

A description of each of the MTX components and the corresponding server functionality they exhibit is described in the following chapters. MTX components are either threads or monitors. An overview is shown in FIGURE 2.

1.5 Design Goals

- Improve interactivity by preventing long protocol requests from locking out other clients.
- Minimize the number of threads.
- Minimize thread context switching.
- Optimize rendering performance.
- Minimize mutex usage. Contended mutexes are expensive, and acquiring mutexes requires automatic update of processor caches to maintain cache coherency.
- Don't be lulled by the 80/20 rule. That is don't generate a design that works for 80% of the cases, but performs poorly for the other less used 20%.
- Avoid optimizations which severely penalize unexpected server use.
- Focus on 8 and 24 bit cfb.

1.6 Design Assumptions

- Framebuffer access is much slower than the ability of a CPU to want to write to it. This bottleneck is even more apparent in a multi-processing environment.
- Support Symmetric Multi-Processing (SMP) platforms with 1 to 10 processors.
- Can modify the existing DIX/DDX interface.
- The server normally has no more than 4 clients executing requests on the server at any one moment.
- There is a small amount of resource sharing. Most Client Input Threads will use objects that are not shared with other threads.

1.7 Threads

A thread is a sequential flow of control. There may be more than one thread executing within a process. Each thread shares the address space of the process with all other threads that are created in the process. A thread has its own execution stack, errno, and thread id. The benefits to using threads are that disjoint sets of code may be executed in parallel while sharing a common code and data address space. Using threads increases the concurrency and interactivity of a process, and allows for more efficient use of multiprocessor architectures.

In order to profit from kernels that support threads and multiprocessors, the MTX server will be targeted to run only on operating systems that conform to the POSIX 1003.4a standard [P1003.4a]. This standard supports

- the creation, control, and termination of threads
- the use of synchronization primitives by threads in a common, shared address space
- per thread data

Users of the X Window System who can not or do not want to use the MTX server will be able to use a single-threaded version of the server.

Note: In this document, the words ***input*** and ***output*** refer to the server's point of view. For example, the Client Input Thread allows the server to take input from the X Client; the Client Output Thread allows the server to send output to the X Client; the Device Input Thread allows the server to take input from hardware devices.

CHAPTER 2 through CHAPTER 7 document the thread components in the MTX Server.

1.8 Monitors

Read and write access to data structures that must be shared by many threads can be effectively managed using the Hoare monitor. This programming construct encapsulates the shared data object in a protective wrapper. The wrapper enforces mutual exclusion by allowing only one thread access to the shared data object at any one moment. The monitor is a global object that advertises all of the public access routines of the object to the current set of threads.

The Hoare monitor is acceptable if the object requires exclusive access. But, it is a poor performer when there are many more readers than writers. Typically, we would like to allow multiple reads to occur concurrently while allowing only one write access at a time.

This solution to the reader-writer problem is handled nicely by the crowd monitor. The crowd monitor has guard procedures that decide when each thread may enter or leave a reader or writer access group. These guard procedures arbitrate access to the protected data object. The crowd monitor determines which group currently has access permission, and queues those threads that must wait.

Monitors will be used where possible to control access to shared server objects.

Locking of MTX server objects is insured by accessing objects only through monitors. CHAPTER 8 through CHAPTER 13 describe this locking strategy and show the details of the MTX monitor components. In those chapters where MTX monitors are described, we discuss both the public interface and the internal design. We think that extension writers will be most interested in the public interface while server developers will be curious about the internal design of the monitor.

1.9 Abbreviations Used

CCT - Client Connection Thread
CIT - Client Input Thread
COT - Client Output Thread
DIT - Device Input Thread
MST - Main Server Thread
SHT - Signal Handling Thread
DE - Device Event Monitor
MOM - Message Output Monitor
POQ - Pending Operation Queue
RDB - Client/Resource Data Base

CHAPTER 2 Main Server Thread

2.1 Overview

The Main Server Thread (MST) manages the global MTX server environment. This thread initializes the different parts of the server and creates the Signal Handling Thread for handling process-wide signals, the Device Input Thread(s) for processing device input, and the Client Connection Thread(s) so that X Clients can establish communications with the server.

If the MST receives a wakeup signal, it cleans up the global MTX server environment and checks to see if the server should reset or terminate. If it should reset, the MST reinitializes the server environment and starts the MTX setup all over again.

The MST is required to insure that server initialization and control is localized in only one area of the server. As such, it is an implementation artifact and does not exhibit the kind of core design functionality that we see in the other server threads. Threads other than the MST, such as the Client Input Thread and the Device Input Thread, exist because they exhibit core functionality.

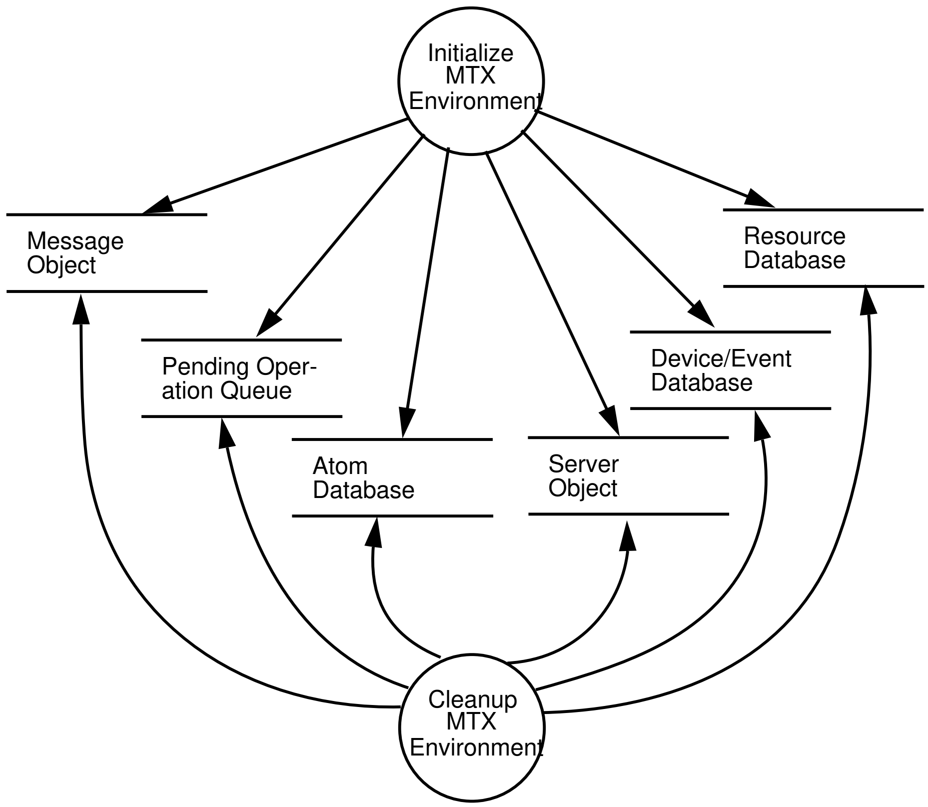

FIGURE 3 Main Server Thread

2.2 Data Objects

Creates the following:

- Resource Database Object
- Message Object
- Device/Event Object
- Pending Operation Queue Object
- Atom Database Object
- Server Object

The Server Object consists of things such as the root window, window table, default font path, default font, scratch GC, scratch stipple, etc.

2.3 Concurrency Issues

When the MTX server is started, it will automatically enter the Main Server Thread and execute the initialization steps in serial order. No other threads will contend for server data objects since no other threads exist at this point. After initializing the server data objects, the MST will create threads for signal handling, device input, and client connection. Then, the MST will sleep until it receives a wakeup signal. Receipt of the wakeup signal indicates that the server should reset itself or terminate.

The MST will create the following threads:

- the Signal Handling Thread
- the Device Input Thread(s)
- the Client Connection Thread(s)

2.4 Internal Organization and Implementation

An overview of the MST is as follows:

```
        mutex_t         ServerMutex;
        cond_t          ServerCondition;

MST();
{
        process the command line arguments (if any)
        initialize MTX global locks
        initialize Message Output Monitor
        block all process-wide signals handled by the server
        create Signal Handling Thread
        lock ServerMutex
        while (1)
        {
                        server initialization
                        wait on ServerCondition to wakeup
                        server cleanup
                        if exit
                                break
        }
        unlock ServerMutex
        exit(0);
}
```

2.4.1 Server Initialization

The MST inner loop will execute the following initialization steps in order:

- Perform OS dependent initialization
- Initialize the Pending Operation Queue Monitor
- Create Well Known Sockets (only once)
- Initialize the Protocol Vectors (only once)
- Initialize the Resource Database Monitor
- Initialize the Serial Number Key
- Initialize the Atom Table object
- Initialize the Device/Event Object
- Initialize output to the Frame Buffer or graphics device objects
- Initialize extension objects
- Initialize the root window object
- Initialize device input object
- Initialize and Create the Device Input Thread(s)
- Initialize the default font objects
- Display the initial cursor and root window
- Initialize Selection objects
- Create the Client Connection Thread(s) (only once)

Note that there is no need for Block and Wakeup Handlers in MTX. Since R5 tries to implement asynchronous activity in a single-threaded environment, these Handlers were needed to give the appearance that the server was waiting. In MTX, the individual threads will block; so these implementation artifacts are no longer needed.

After the server is initialized, the MST will sleep on a pthread signal. This signal will be triggered by the exit of the last CIT when it detects that there are no more clients connected to the MTX server, or when the server receives a UNIX signal.

2.4.1.1 Initialize Input Device Objects

MTX initializes the Device Input Thread (DIT) state and then, just as in R5, the MTX server will make a call to the DDX function *InitInput* which calls *AddInputDevice* for each core device. *AddInputDevice* creates the device object and registers the device procs for the passed device. The device procs are written by the DDX implementor. For example, in the sample implementation for the Luna88k, the device procs are called *OmronKbdProc* and *OmronMouseProc*.

Next, the server will call *InitAndStartDevices*. For each device, the device proc that was registered for the device object is called with DEVICE_INIT to finish the device specific initialization, and then called with DEVICE_ON to setup the proper file descriptor.

After *InitAndStartDevices* returns, MST calls the DDX routine *CreateDeviceInputThread*. *CreateDeviceInputThread* is used to create the DIT. This mechanism allows the DDX implementor to create multiple DITs if that is the goal. However, at this time the sample implementation of *CreateDeviceInputThread* creates only one DIT.

2.4.2 Server Control: Wait On Signal to Wakeup

After completing server initialization, the MST waits on a condition variable until it receives a signal indicating that the cleanup phase should begin. The condition will be signaled when any of the following occur:

- The server receives a SIGINT or SIGTERM signal
- The server receives a SIGHUP signal (typically sent by XDM)
- No clients remain connected to the server following a client disconnect

Signals are handled by the Signal Handling Thread (see CHAPTER 7).

2.4.3 Server Cleanup

When the MST has been signalled, it cleans up the global MTX server environment in the following way:

- Terminate the Device Input Thread, wait for it to exit
- Initiate shutdown of all CITs still attached to the server
- Wait for all CITs to exit
- Close extensions
- Clean up the Resource Database
- Free all global objects
- Close down all input and output devices
- Free all font objects

If the server is to TERMINATE, then the MST performs device specific cleanup and exits. If the server can be RESET, the MST cycles back to the above initialization steps and repeats the environment setup for the server, except that the Client Connection Thread is not recreated. The CCT is never terminated. New connection requests will pend while the server is being reset (see Section 2.4.5).

2.4.4 Extensions

Extensions are initialized during the server initialization phase.

NOTE: Command line options may be used to specify dynamic loading of extensions. In this way, the extension initialization code (*InitExtensions*) can determine which extensions should be loaded. This saves space in the raw server executable if shared libraries are used.

2.4.5 How are other Threads Affected

Client Connection Thread: The CCT must not allow new connection requests to be processed while the server is resetting or terminating. The MST holds the ServerMutex at all times during initialization and cleanup, only releasing the mutex when it eventually sleeps on the ServerCondition. The CCT will acquire the same mutex after detecting a new connection request but before processing it. Thus, the CCT can only process new connections when the MST is sleeping on the ServerCondition (see CHAPTER 3).

This also assures the MST will not begin a cleanup phase while a connection request is being processed.

Client Input Thread: The MST should be signaled to initiate a server reset when a disconnecting client will result in no clients being connected to the server. Just prior to thread termination, a CIT will check to see if other CITs remain active in the server. If not, the terminating CIT will signal the MST to initiate a reset (see CHAPTER 4).

Device Input Thread: The DIT will be signaled by the MST to shutdown during a server reset or termination. The MST will wait on the ServerCondition for a signal from the DIT that it has shutdown (see CHAPTER 6).

Signal Handling Thread: The signal handling thread will signal the MST to initiate a server reset or termination upon receipt of certain UNIX signals. A global variable will indicate whether the server is to reset or terminate (see CHAPTER 7).

2.4.6 R5 versus MTX

The functionality of the Main Server Thread is similar to that found in *main* in *main.c*. In the R5 implementation, the *dispatch* exists so that new client connections, new protocol requests, and new device input can be arbitrated fairly by the server. In the MTX server, however, these functions are broken into independent flows of control. Therefore, the *dispatch* loop has been removed from the MTX server.

CHAPTER 3 Client Connection Thread

3.1 Overview

The Client Connection Thread (CCT) receives a request by an X client to connect to the X server. New client connections are processed by a single CCT as shown in FIGURE 4. Upon accepting a client connection, the CCT immediately creates a Client Input Thread to continue the validation processing. This CIT validation includes all new connection authentication and authorization

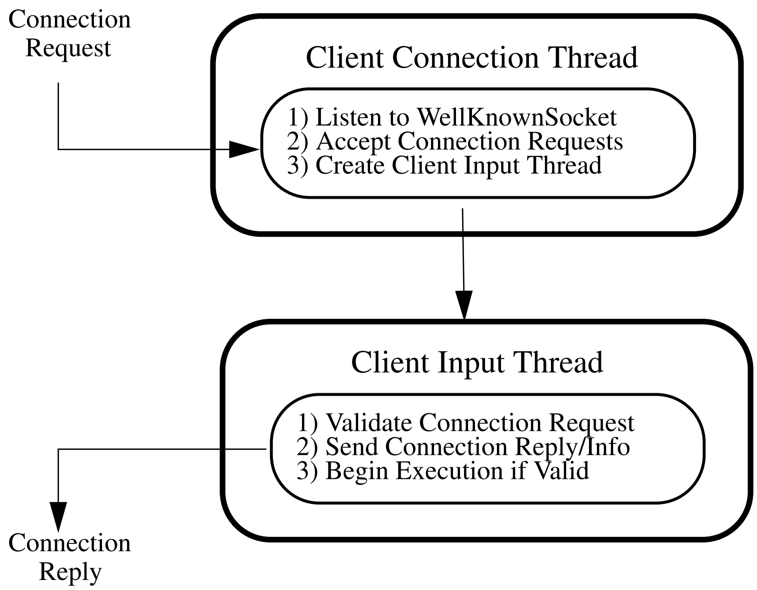

FIGURE 4 Client Connection Thread

3.2 Data Objects

Each Client Connection Thread will only access the OS Connection object via the ConnectionMutex. The OS Connection object includes:

- ConnectionMutex
- numberOfClients
- Transport Layer connection info (such as HOST)
- Some access control info

3.3 Concurrency Issues

The CCT cannot process connection requests while the Main Server Thread is resetting or terminating the server. After detecting a connection request, the CCT will obtain the ServerMutex lock before processing. The MST holds this lock at all times during initialization and cleanup, thus assuring that the CCT will not process connection requests during this phase.

The CCT creates the:

- Client Input Thread

3.4 Internal Organization and Implementation

3.4.1 Client Connections

The CCT listens for new connections on the well-known ports. When a new connection is detected, the CCT accepts the connection and creates the CIT so that a separate connection unique to the requestor can be created. The CCT does not perform client specific connection validation; this is left to the newly created CIT.

Since the CCT only validates the connection type, it would be possible to have CCTs that are connection type specific. For example, the default CCT handles socket connections, but one could also have a CCT that performed shared memory connections. In the shared memory CCT, only shared memory type operations are validated before creating the CIT. In this way, the CIT is not dependent on the type of connection existing between the server and the X client.

```
CCT()
{
	lock ServerMutex
	while(1)
	{
		unlock ServerMutex
		read well known port
		lock ServerMutex
		if valid connection type
			store client connection info
			create CIT
		else
			refuse connection
	}
	unlock ServerMutex
}
```

In addition to decoupling the connection validation from the client validation, we push the validation of the client requesting a connection out of the CCT so that the CCT does not block during the validation handshake.

3.4.2 Server Initialization and Control

The CCT(s) is created once by the MST during server initialization and never destroyed, except if the server terminates. The MST takes this into account when resetting the server.

3.4.3 Client Initialization and Control

The CCT authenticates the connection specific information for the new client. The CCT does not check the authority of the client to be connected to the server. This is performed by the Client Input Thread.

This two level method of validating the client unbinds the connection specific checks from the client specific checks, and allows the CCT to continue accepting connections without being dependent on the actual validation of the client. If the connection fails, the CCT gives the reason and closes the network connection.

3.4.4 R5 versus MTX

The R5 server checked for new client connections, along with handling device input and protocol requests, in *WaitForSomething*. This module has been eliminated because these three functions do not require serial execution.

The CCT uses *EstablishNewConnection* to complete the connection type semantics and prepare the CIT to perform client validation. The top level structure chart is shown in FIGURE 5 and the *EstablishNewConnection* procedure in FIGURE 6.

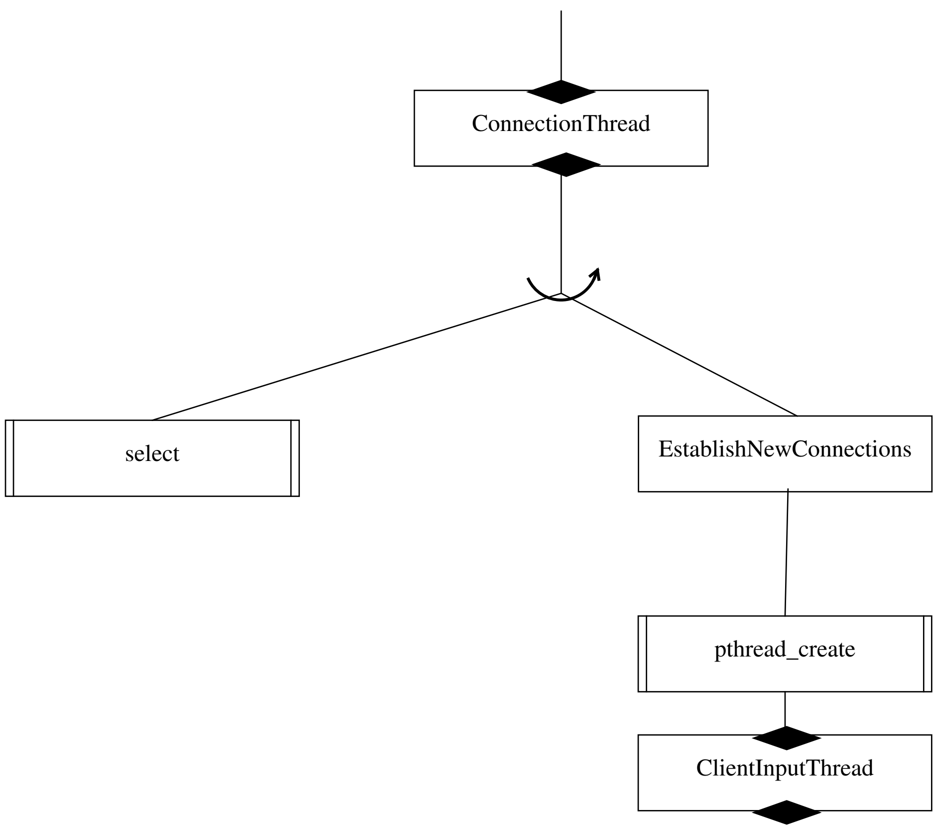

FIGURE 5 CCT Top Level Structure Chart

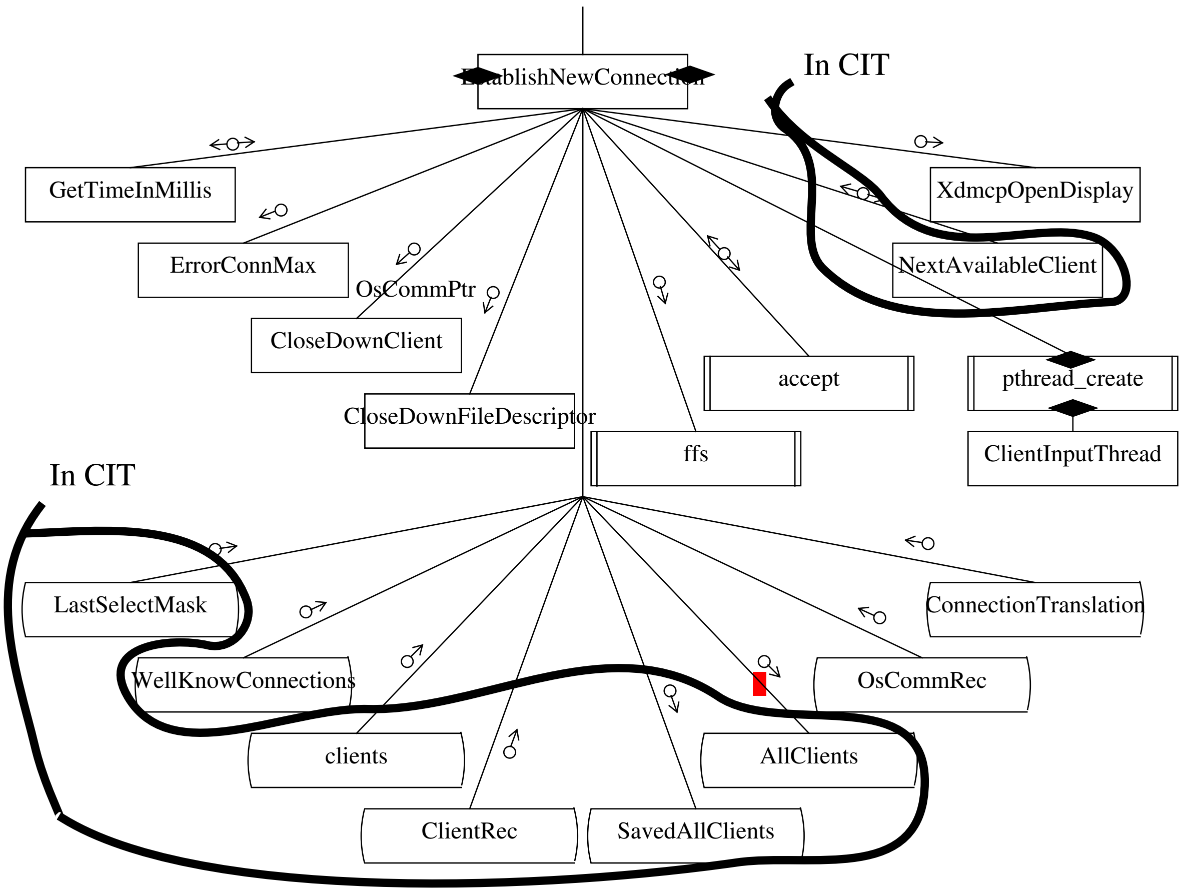

FIGURE 6 CCT Establish New Connections Structure Chart

CHAPTER 4 Client Input Thread

4.1 Overview

The Client Input Thread (CIT) processes client protocol requests for the connected client only. There exists a CIT for each client connected to the MTX server. The function of this thread (see FIGURE 7) is to process requests for its assigned client while adhering to the atomicity and serial constraints for client event and reply notification.

When the CIT is first created, it validates the new client connection. The CIT accepts the client connection if successful and adds the client information to the client database.

After validation, normal CIT operations will begin.

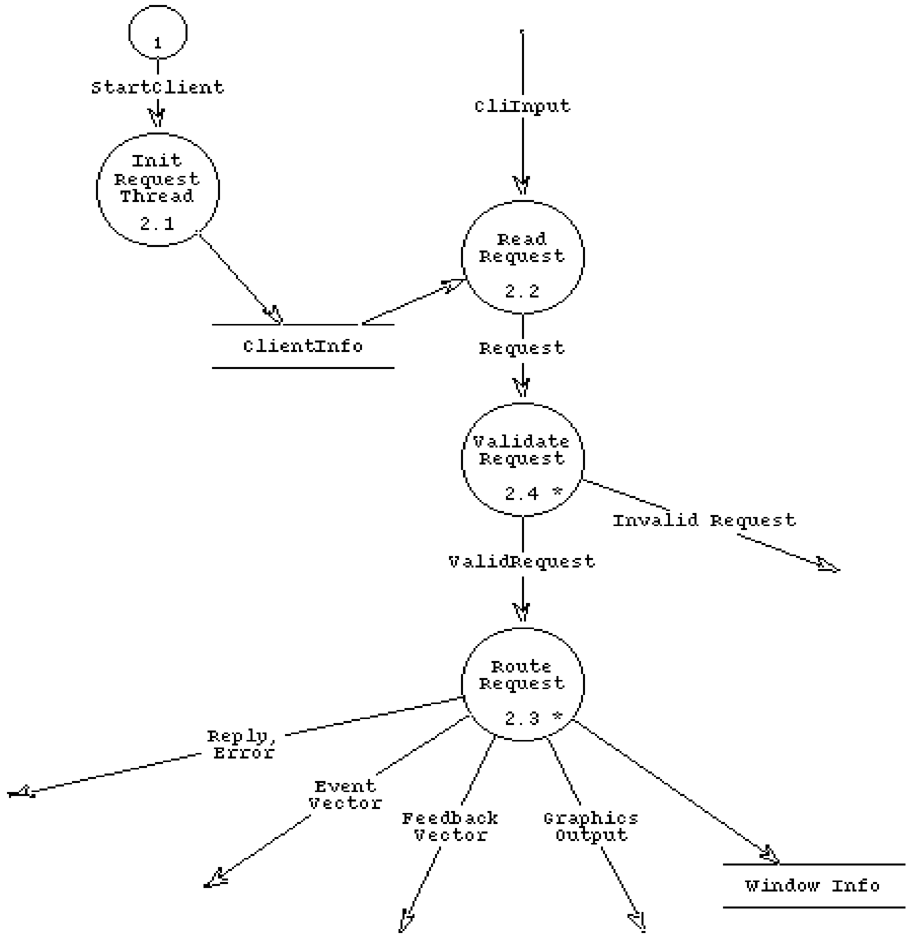

FIGURE 7 Client Input Thread

4.2 Data Objects

The CIT may access any data object in the server if it has a handle to that object. However, the CIT usually refers to objects that only it creates and manipulates (a notable exception is the window manager CIT). This "locality of access" means that for most accesses, the CIT only needs to refer to it's own objects in the Client Resource Table. Those objects typically touched by a particular class of protocol requests is shown in FIGURE 8.

WINDOW
1 CreateWindow
2 ChangeWindowAttributes
3 GetWindowAttributes
4 DestroyWindow
5 DestroySubWindows (11)
6 ChangeSaveSet
7 ReparentWindow (8,10)
8 MapWindow
9 MapSubWindows
10 UnmapWindow
11 UnmapSubWindows
12 ConfigureWindow
13 CirculateWindow
14 GetGeometry
15 QueryTree
40 TranslateCoords

ATOM
16 InternAtom
17 GetAtomName

PROPERTY
18 ChangeProperty
19 DeleteProperty
20 GetProperty
21 ListProperties
114 RotateProperties

SELECTION
22 Selection
23 GetSelectionOwner
24 ConvertSelection

EVENT
25 SendEvent
35 AllowEvents

POINTER
26 GrabPointer
27 UngrabPointer
30 ChangeActivePointerGrab
38 QueryPointer
39 GetMotionEvents
41 WarpPointer
105 ChangePointerControl
106 GetPointerControl
116 SetPointerMapping
117 GetPointerMapping

BUTTON
28 GrabButton
29 UngrabButton

KEYBOARD
31 GrabKeyboard
32 UngrabKeyboard
33 GrabKey
34 UngrabKey
44 QueryKeymap
100 ChangeKeyboardMapping
101 GetKeyboardMapping
102 ChangeKeyboardControl
103 GetKeyboardControl
104 Bell
118 SetModifierMapping
119 GetModifierMapping

SERVER
36 GrabServer
37 UngrabServer

FOCUS
42 SetInputFocus
43 GetInputFocus

FONT
45 OpenFont
46 CloseFont
47 QueryFont
48 QueryTextExtents
49 ListFonts
50 ListFontsWithInfo (47)
51 SetFontPath
52 GetFontPath

PIXMAP
53 CreatePixmap
54 FreePixmap

GC
55 CreateGC (56)
56 ChangeGC (46)
57 CopyGC (46)
58 SetDashes
59 SetClipRectangles
60 FreeGC

GC AREA
61 ClearArea
62 CopyArea
63 CopyPlane

GC 2D
64 PolyPoint
65 PolyLine
66 PolySegment
67 PolyRectangle
68 PolyArc
69 FillPoly
70 PolyFillRectangle
71 PolyFillArc

IMAGE
72 PutImage
73 GetImage

TEXT
74 PolyText8
75 PolyText16 (74)
76 ImageText8
77 ImageText16 (76)

COLORMAP
78 CreateColormap
79 FreeColormap
80 CopyColormapAndFree (78,89)
81 InstallColorMap
82 UninstallColormap (81)
83 ListInstalledColormap
84 AllocColor
85 AllocNamedColor (84)
86 AllocColorCells
87 AllocColorPlanes
88 FreeColors
89 StoreColors
90 StoreNamedColor (89)
91 QueryColors
92 LookupColor

CURSOR
93 CreateCursor (73)
94 CreateGlyphCursor (73)
95 Free Cursor
96 RecolorCursor
97 QueryBestSize

EXTENSION
98 QueryExtension
99 ListExtensions

SCREENSAVER
107 SetScreenSaver
108 GetScreenSaver
115 ForceScreenSaver (1,8)

ACCESSCONTROL
109 ChangeHosts
110 ListHosts
111 SetAccessControl

CLIENT
112 SetCloseDownMode
113 KillClient

MISC
127 NoOperation

FIGURE 8 Protocol Requests categorized by Object

Any CIT may potentially access any of the object categories listed in FIGURE 9. Because the Client Input Thread executes X Protocol requests, object usage for these threads should be based on the type of request. FIGURE 9 shows each protocol request and the data object categories used by that request. Only the major DIX and OS data objects are used. Each data object category can be accessed in the following ways:

- **R** - read only
- **W** - read/write
- **D** - delete
- **C** - Create
- ***** - **W**,**D**,**C** can all occur

	Data Object Category														
Protocol Requests Category	**Client**	**Color**	**Cursor**	**Device**	**DIXFontInfo**	**EventMasks**	**GC**	**OSComm**	**OSFontInfo**	**Pixmap**	**Property**	**Screen**	**Selection**	**Window**	**AtomTable**
Window	W		W	R		W				R		W		*	
Atom	W														W
Property	W		D								*			R	
Selection	W												*	R	
Event	W		R	R										R	
Pointer	W		W	W						R		R		R	
Button	W		*	*								R		W	
Keyboard	W		W	*						R		R		W	
Server								C							
Focus	W			R						R				*	
Font					R		R		*						
Pixmap	W									C		R		R	
GC	W						*			*		R			
GCArea	W						W			W		R			
GC2D	R						R			R					
Image	R						W			R		R			
Text	R				R		W			R					
Colormap	W	*								R		R		R	
Cursor	W		*		R					R		R			
Extension	R														
ScreenSaver	W	R					*			W		W		W	
AccessContrl	R														
Client	*			W									W		
Misc															

FIGURE 9 Data Object Usage by Protocol Requests

4.3 Concurrency Issues

Each Client Input Thread may contend with other CITs and the Device Input Thread (DIT) for access to the Device/Event Object. All CITs and the DIT will use the Device/ Event Monitor to arbitrate access.

All CITs may contend for resources in the Client Resource Table. The Client Resource locking strategy (see CHAPTER 9), will control access to these contended resources.

All events, replies, and errors generated as a side-effect of the protocol request are buffered up into Message Objects. The Message Object Monitor allows the CIT to send messages directly back to the X client, or by creating a Client Output Thread for batch sending. The creation of COTs allows the CIT to be independent of how long the X client may wait before reading on its end.

This thread must share the Message, Device/Event and Client Resource Table objects with other CITs and the DIT.

The CIT writes to Message Objects that may be used by a COT.

The last remaining CIT will signal the Main Server Thread to wakeup before it exits.

4.4 Internal Organization and Implementation

4.4.1 General Thread of Control

```
CIT()
{
    /* the CIT has just been created by the CCT */
    lock ConnectionMutex
    validate new X client
    if (not valid client connection)
        unlock ConnectionMutex
        return failure status to X client
        exit thread
    numberOfClients++;
    unlock ConnectionMutex
    allocate local message buffer

    while(1)
    {
        Read request block from client
        if read fails
            break
        for <each request in the read block>
        {
            increment client sequence number
            get RDB locks for this specific protocol request
            insert element in POQ and wait until no conflicts
```

```
                if <request requires "come-up-for-air">
                        FlushAllMessages()

                execute request, buffering all events/replies/errors
                        (client output messages) locally
                if <any events/errors/replies generated by request>
                        transfer local buffer entries to global buffer

                remove node from POQ list
                release RDB locks
            }
            FlushAllMessages()
        }

        Close down client connection
        if last CIT
            if terminateAtReset
                signal MST to terminate the server
            else
                signal MST to reset server
            signal ServerCond in case MST waiting in KillAllClients
        FlushAllMessages()
}
```

Render related protocol requests are designed so that they check periodically (for example, every time they enter *FillSpans*) to see if they should pend for another server operation that needs to manipulate their render window. When that operation completes, the render protocol request continues. This feature is called "coming-up-for-air". It allows, for example, a long render request to be interrupted in order to move a software cursor across the render window, or have the window manager repaint the border windows. It is an optimization that enhances user interactivity. This optimization may prove useful for extension requests, provided that it can be implemented without violating any atomicity requirements of a request.

4.4.2 Request Processing

The CIT will block until there is a pending client request. This thread handles the byte ordering of the network connection to the client. The request is processed similar to the function pointer mechanism used in the *dispatch* loop for *(* client-requestVector[MAJOROP])(client)*.

These protocol requests are grouped into the functional categories described in FIGURE 8, which lists the requests by the type of objects that they access. In the diagram, the protocol request codes are also listed. Some codes invoke other codes; in this case the invoked codes are parenthesized. For example, ReparentWindow(7) does most of MapWindow(8) and UnMapWindow (10).

All CITs will be able to process any protocol request including any pre-initialized extensions. Since each protocol request accesses both shared and private resources, locking policies will be invoked to protect multiple threads from corrupting shared resources.

For shared resources, the Client/Resource Locking strategy (see CHAPTER 9) is the arbitration policy. The CIT locks objects in the Client/Resource Database and inserts the protocol request on the Pending Operation Queue (POQ).

If a CIT wishes to gain access to event related objects (such as the event mask, GrabRec, or DeviceIntRec), the Client Input Thread will request access through the Device/Event Monitor (see CHAPTER 11). The Device/Event Monitor will insure exclusive access to the event related objects and block other requesting threads to enforce serial access to the event related objects.

As noted above, render protocol requests will have the ability to come-up-for-air.

All events that are generated as side effects of processing the protocol request will be processed by the Device/Event Monitor. Events, replies, and errors from a protocol request are then wrapped into messages via the Message Output Monitor. Errors and replies that originate in the Client Input Thread will be sent directly to the corresponding X Client. Events targeted for multiple clients will be buffered for batch processing by a Client Output Thread. In the MTX sample implementation, the COT is created dynamically within the Message Output Monitor (see CHAPTER 12) as the result of a flush, potentially initiated from a CIT. The MOM could be altered to allow the option of dynamic or static COTs, but only dynamic COTs are provided in the sample implementation.

The CIT will also handle client shutdown activities. When a client dies or is killed, the appropriate CIT will free resources from the Client/Resource Table and cleanup local data structures. If the CIT represents the last client connected to the MTX server, then it will notify the Main Server Thread to wakeup and reset the server.

4.4.2.1 GrabServer

A CIT cannot continue to process requests while another CIT has a pending GrabServer. The CIT will block on the POQ (see CHAPTER 10) until the grabbing CIT performs an UngrabServer request.

4.4.3 Client Output Delivery

Client Output, which includes events, errors, and replies, can be generated by the protocol request back to the X client. All protocol requests can return errors. Some protocol requests can generate. events and replies. See [SG90] for a more detailed description.

The Device Input Thread can also generate events for an X client. In fact, multiple CITs and the DIT may generate events to the same X client at the same time. All events generated by a single request in the CIT, or device input in the DIT, must be queued for delivery indivisibly with respect to events generated by other threads. See CHAPTER 12, the Message Output Monitor, for details about how the Client Output atomicity requirements are met.

See CHAPTER 5 for a description of how the COT arranges for client output delivery.

4.4.3.1 Flushing Output to Clients

Flushing of all client messages occurs at the following locations of the server:

- at the end of the CIT's request block loop
- after the CIT has executed a client shutdown but before the thread exits
- after processing a device input event from DIT
- inside request loop of CIT but prior to arbitrarily long requests

4.4.4 Server Initialization and Control

The CCT creates the CIT during the client initialization phase. When the MST is told to reset, it will wait until the last CIT has exited before proceeding with the cleanup phase.

4.4.5 Client Initialization and Control

A CIT is created by the Client Connection Thread when an X Client wants to connect to the server. The CCT will validate the connection by running the *EstablishNewConnection* functionality. After the new connection is established, the CIT is created.

The CIT will then run the *InitialConnection* and *EstablishConnection* functions. These requests are used to validate the connection and trade server information with the X Client. The new client sends a connection setup packet that contains the byte ordering for its host machine and authorization information.

Client specific authentication of the client connection is performed by the CIT as soon as it is created by the CCT. This frees the CCT to service other new client connections and should speed the process of starting up an X session.

After successfully validating the connection, the X client is considered to exist on the server side. The CIT then enters a continuous loop of reading protocol requests and operating on those requests. It only exits the loop when client termination is requested either voluntarily by the client, or involuntarily as a result of a server exception.

4.4.6 DDX Layer

If there is output to be sent to any devices, a CIT will manage that activity through the DDX layer. This includes writing to the graphics output device and generating feedback for the feedback devices (i.e. led and bell). Writing to the graphics output devices is a frequent operation and could generate an unacceptable number of thread context switches if this functionality were placed in a thread other than the Client Input Thread.

4.4.7 ScreenSaver

A CIT will control the screen saver through the GetScreenSaver, SetScreenSaver, and ForceScreenSaver protocol requests (see CHAPTER 15).

4.4.8 XDM Interface

The initial connection by the display manager client to MTX will create a CIT dedicated to XDM. The XDM CIT is used to communicate XDMCP requests to the server.

4.4.9 FontServer Interface

The X server side of the Font Server (FS) will be adjusted so that only the CIT requesting information from the Font Server will block on the FS connection. The X server will not need to receive Suspended status from the FS, nor will it need the work in progress queues to handle pending FS requests.

4.4.10 Extensions

Extensions should model the approach used by core requests to lock server objects.

4.4.11 R5 versus MTX

In the R5 server, the *select* fires when one of the file descriptors detects an X Client request. Each file descriptor is correlated with a connected client via the *AllClients* mask. The index into the mask is used to find the correct client record. The client record points to the input and output OS Comm records for the client. The protocol request is read from the input OS Comm record. The *dispatch* loop uses the major operation code of the request as an index into the request vector. This entry in the request vector is a pointer to the correct protocol function that will process the X Client's request.

The MTX server localizes the processing of client requests to a CIT. Therefore, there is no need for all of the various client connection masks that are found in R5's *connection.c* file. The CIT does a blocking read on the client's request socket; thus removing the dependence on the slow *select* call. Once the CIT detects a request, it does not have to use or manipulate any complicated masks to determine the client record or the client id. The CIT can immediately jump to validation of the request.

A CIT will have one input buffer and not manage a pool of buffers as currently exists in the R5 server. This approach allows the CIT to avoid extra locks in a performance critical path of the server and trades-off the possibility of returning an input buffer to the system for clients that rarely issues requests.

FIGURE 10 shows that the top level of the Client Input Thread is a subset of the old *dispatch* code.

FIGURE 11 through FIGURE 12 shows a logical division of the requests into protocol request categories with the create window request of the window category used as an example. This example continues in FIGURE 13 through FIGURE 14.

The create window example is typical of how each protocol request *XXX* is handled. The top layer of the Proc*XXX* module write locks the row of the Client Resource Table corresponding to the passed client. Generally, most of the processing is performed in the *XXX* module. The XXX module may access the Message Object and the Device/Event Object to send events or generate grabs. Access to the Device/Event Object is guarded by the DeviceEventMutex. Each batch of event related calls is preceded by a lock of DeviceEventMutex and then succeeded by the paired unlock of DeviceEventMutex.

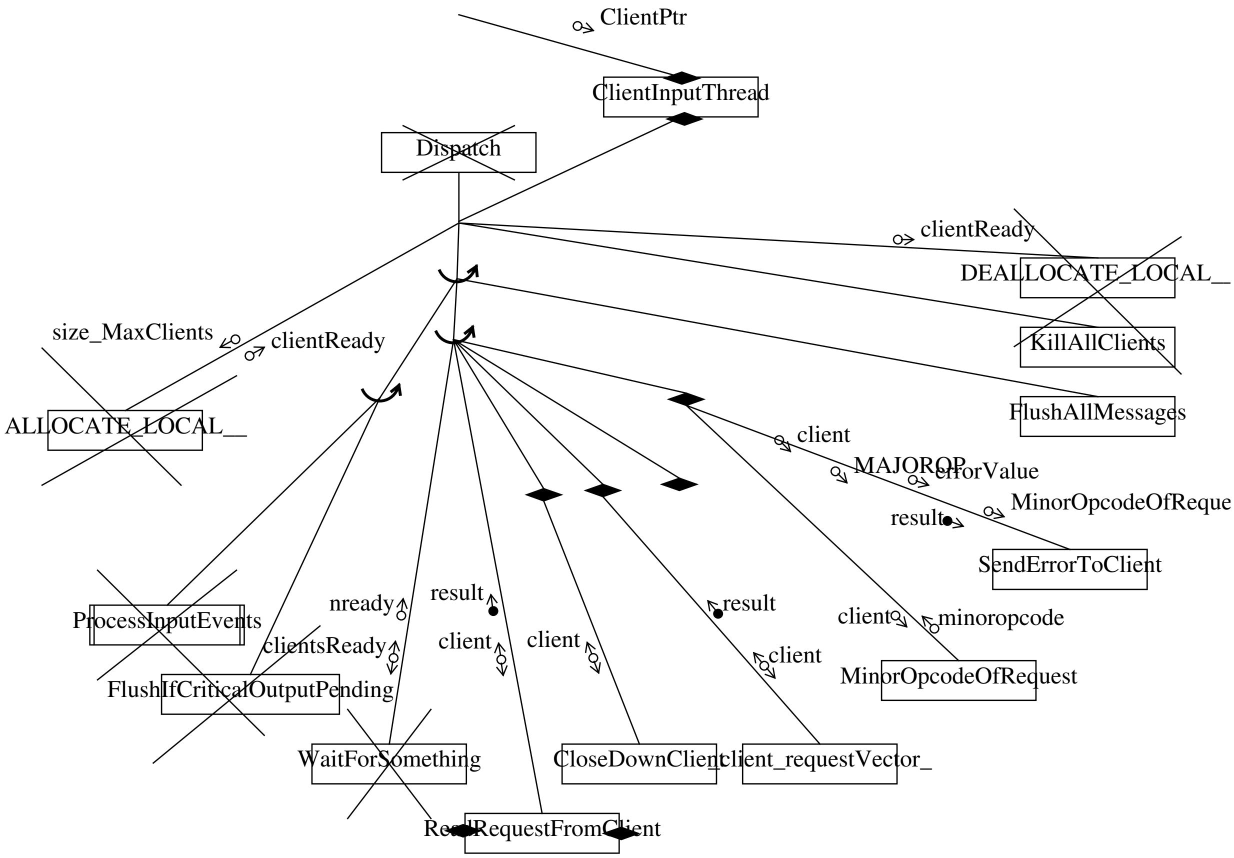

FIGURE 10 CIT Top Level Structure Chart

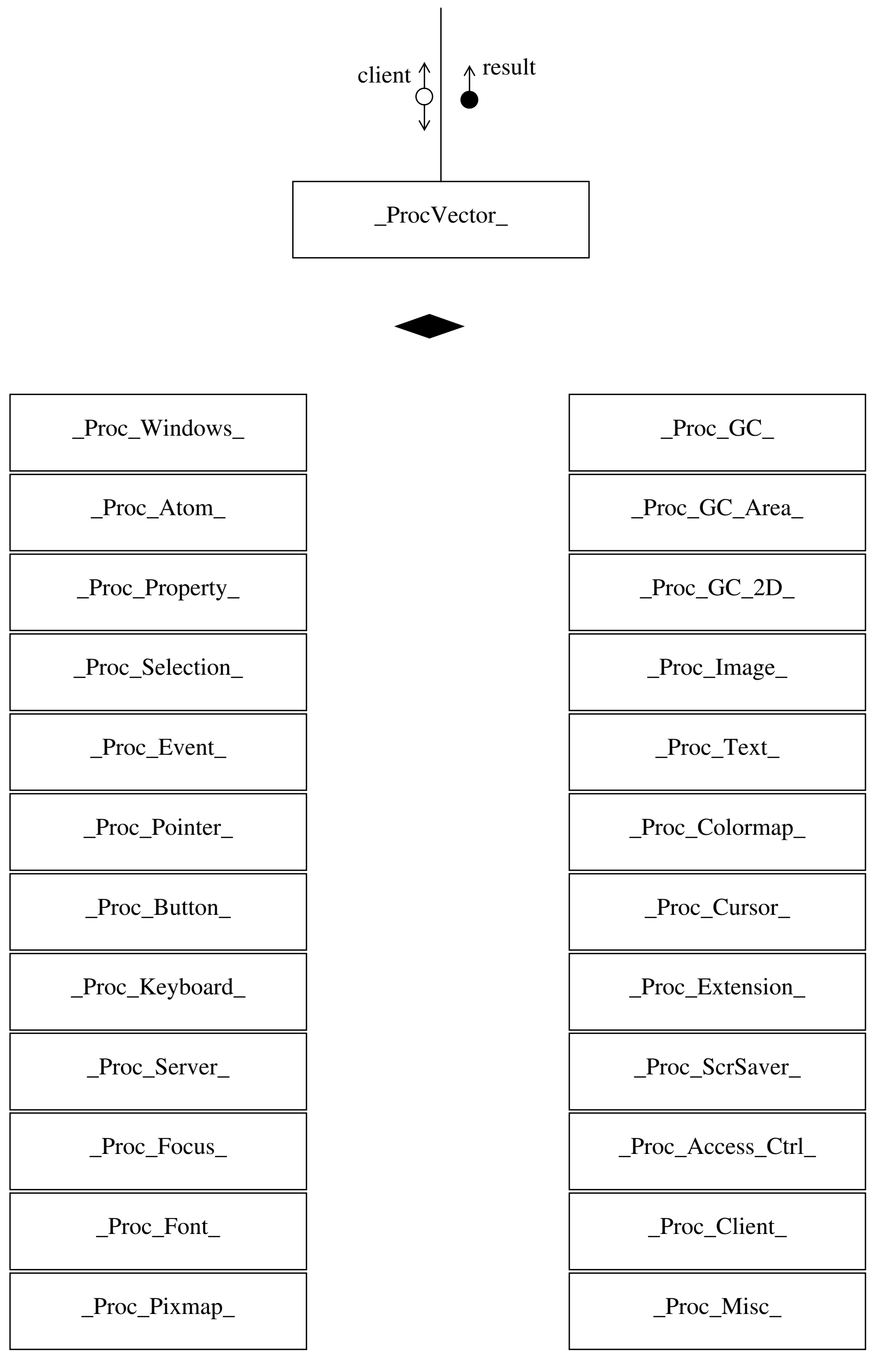

FIGURE 11 CIT Proc Vector Logical Structure Chart

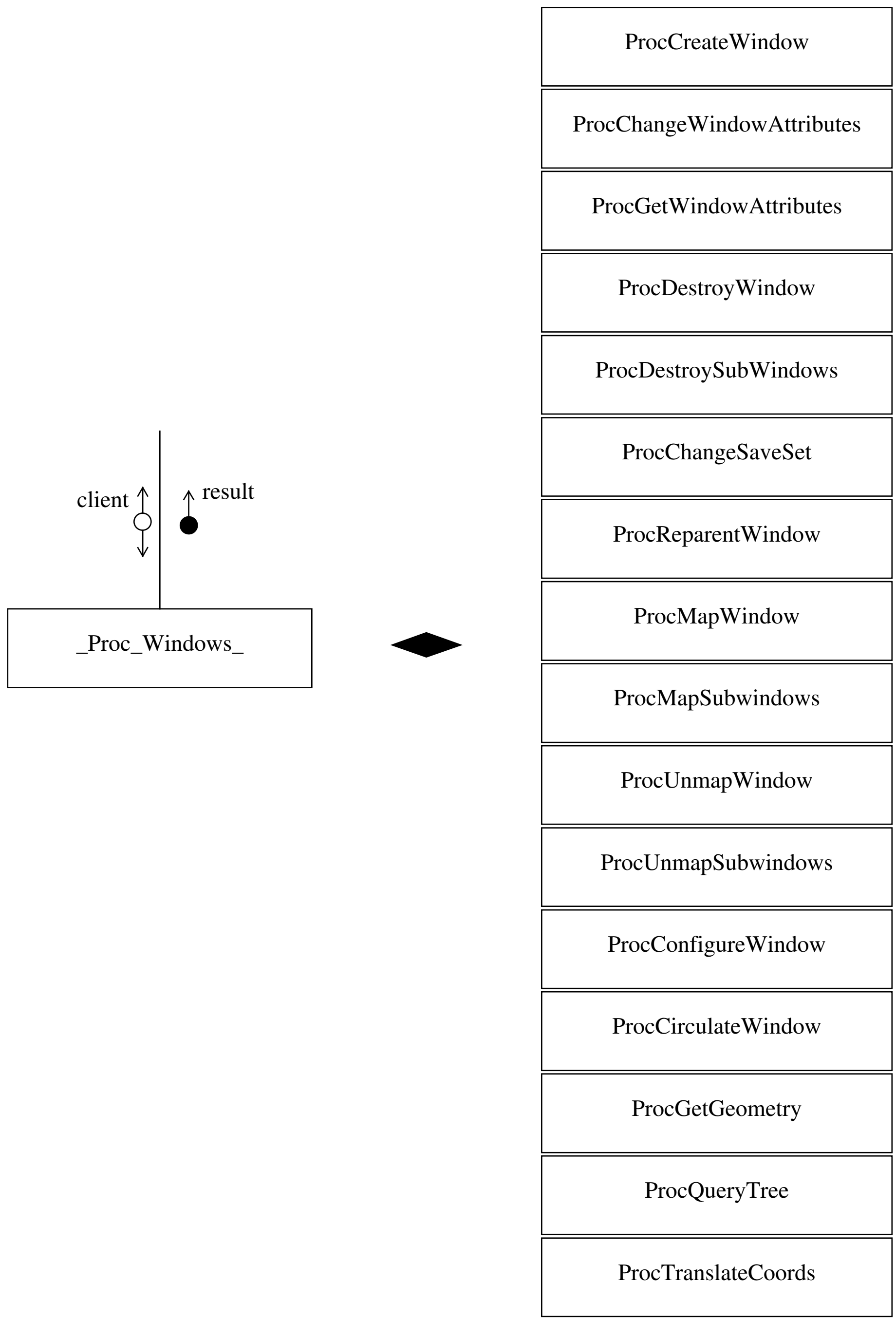

FIGURE 12 CIT Proc Windows Logical Structure Chart

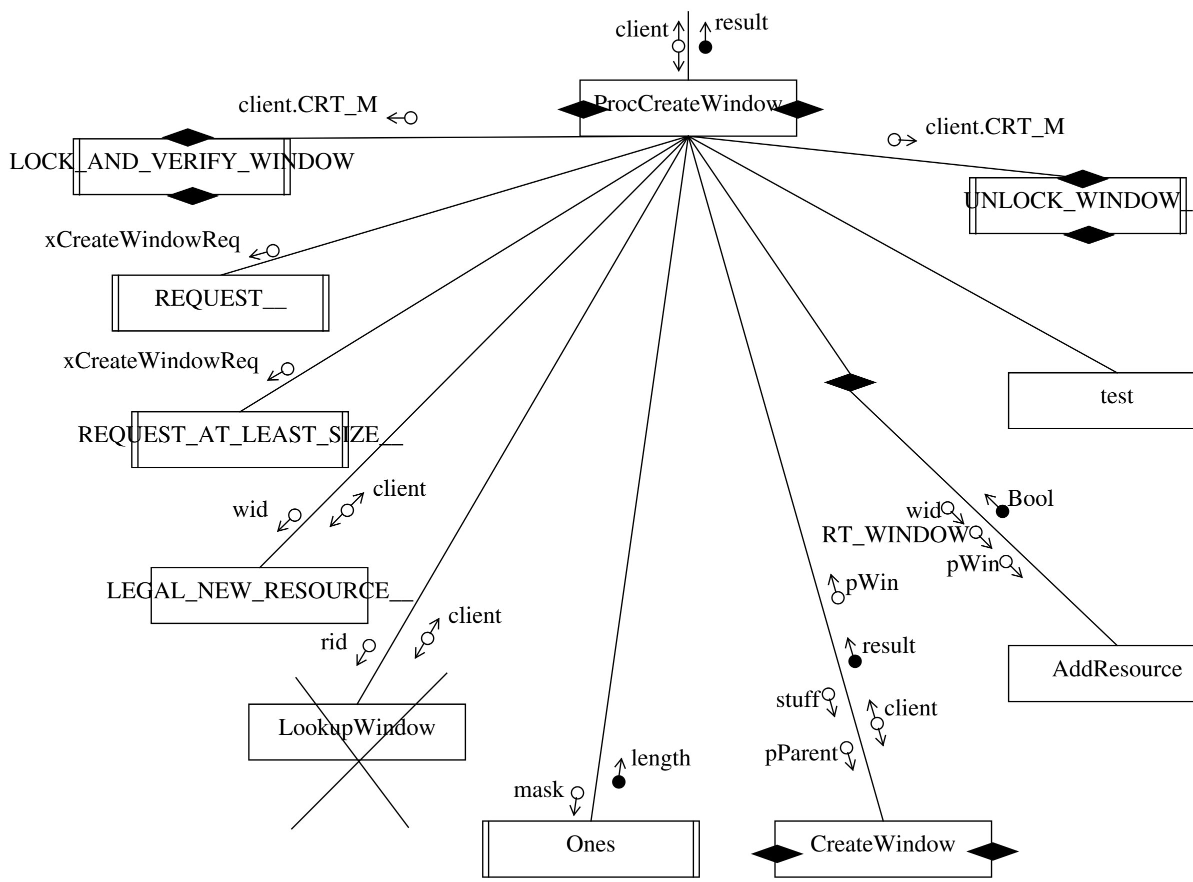

FIGURE 13 CIT ProcCreateWindow Structure Chart

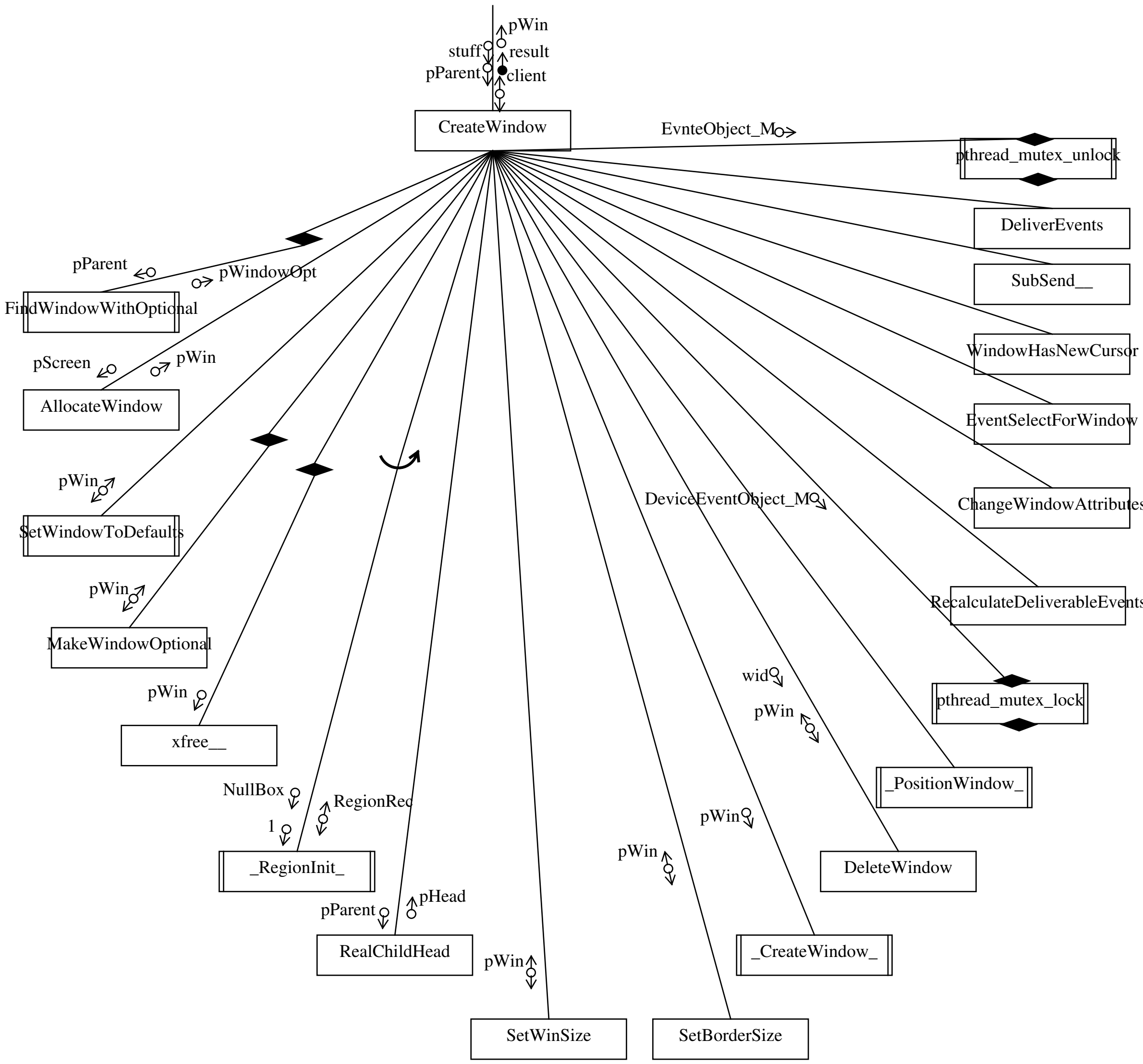

FIGURE 14 CIT CreateWindow Structure Chart

CHAPTER 5

Client Output Thread

5.1 Overview

Route messages from the server to the client.

The Client Output Thread (COT) thread (see FIGURE 15) is created within the Message Output Monitor (see CHAPTER 12) when a message flush occurs. The flush will have been initiated by a Client Input Thread or the Device Input Thread when these threads must send messages to X Clients.

A COTs sole purpose is to gather messages for a target client that have already been generated and queued, synchronize with other threads that may be trying to write to the same target client, and write the message data to the target clients socket.

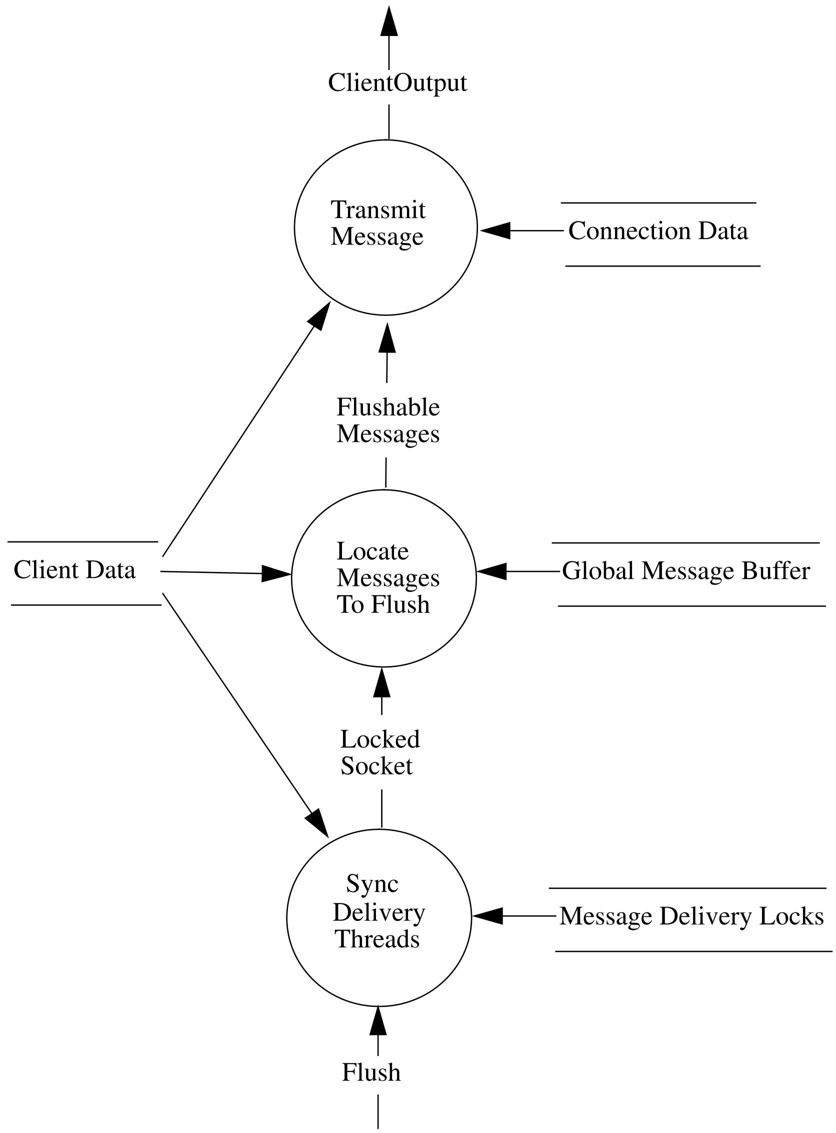

FIGURE 15 Client Output Thread

5.2 Data Objects

The Client Output Thread will use:

- the connection information for the client
- client specific data
- the global message buffer built by the producer thread
- message delivery locks, for synchronization

5.3 Concurrency Issues

COTs are created only when needed and terminate after completing the delivery task. When the server communicates with clients via blocking I/O, it is required that the server avoid situations where a CIT or DIT might block because a certain client is not reading its messages. So, a COT was created for the sole purpose of delivering a set of messages to target client and then terminating. With this mechanism, a CIT or DIT will not block due to a client who has a full socket buffer.

There is one exception where COTs are not created for delivering messages. If a CIT is delivering messages directly back to the client to which it is associated, the CIT can deliver the messages. In this case, it is considered reasonable for a CIT to block if the client is not reading from its socket buffer. Note that round trip requests will always have a reply delivered from the CIT that generates it. This optimization saves a thread creation and context switch.

5.4 Internal Organization and Implementation

5.4.1 General Thread of Control

```
COT()
{
        look at implicit MessageDeliveryLock information
        if <thread already waiting for same implicit lock>
                thread exit
        obtain implicit MessageDeliveryLock

        lock global message buffer
        locate sublist of flushable messages
        if <sublist is not empty>
                remove sublist from list of all messages pending for client
                unlock global message buffer
                build IOV array locally
                writev to socket
                free sublist
        else
                unlock global message buffer

        release implicit MessageDeliveryLock
        thread exit
}
```

It is possible that more than one COT for the same client may exist at the same time. One of them will grab the implicit MessageDeliveryLock for the target client, one will wait for the currently executing COT to finish, while all others will realize that any current messages will eventually be flushed by the waiting thread and merely terminate without doing anything.

Note that if a client is not reading from its socket, it is possible that a COT will block when it issues a writev(2). Messages destined for that client will begin to accumulate in the global message buffer. A mechanism is built into the Message Output Monitor that will try to detect this situation when memory cannot be allocated and will try to reclaim these messages (see CHAPTER 12).

The normal case for CIT generated events/replies/errors is that the CIT itself will be able to flush to the client and a COT will not have to be created.

Since most protocol requests will send directly to the socket, and DIT events happen relatively infrequently, creating COTs on the fly is a good idea because it will minimize thread management. We don't want idle COTs sitting around taking up precious kernel structures. Also, with fewer COTs, the chance of a cache hit to find a COT is greater. It is believed that many threads based kernels can create threads very fast, perhaps even faster than it takes a thread to context switch into memory.

5.4.2 R5 versus MTX

In R5, client output delivery is handled by the protocol request or device input code at the moment it needs to deliver using *WriteToClient*(). In MTX, we decouple the delivery of messages for the CIT, and we will decouple the delivery of events for the DIT. So, the *WriteToClient* interface will be changed for producer threads to reflect the algorithm described in CHAPTER 4.

Since the COT uses many of the Message Object Monitor features, please see CHAPTER 12 for more details on message delivery.

CHAPTER 6 Device Input Thread

6.1 Overview

The Device Input Thread (DIT) waits for device input and routes device events to selected Client Output Threads.

There will be one Device Input Thread to handle all device input to the server, although the ability to create multiple DITs will not be prohibited. In R5, device input from the mouse, keyboard, trackball, etc. currently executes in the *ProcessInputEvents* function. This function has device independent and device dependent code based on the type of device. The Device Input Thread (see FIGURE 16) will accept input from one or more registered devices. This thread will process the device input, format the correct device events, and deliver the device events through Client Output Threads.

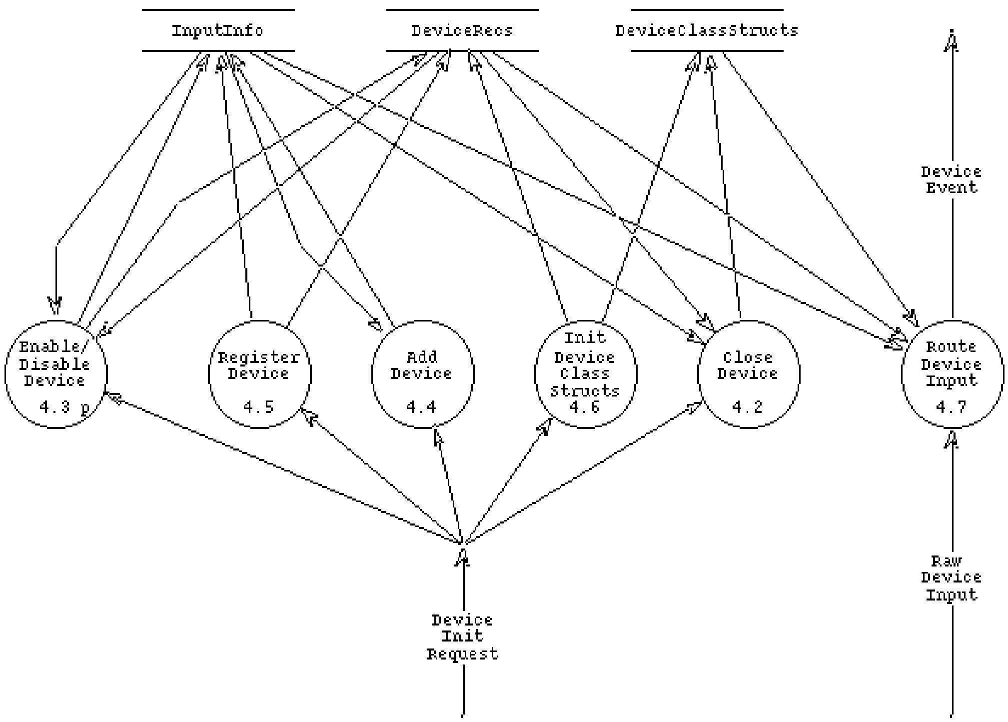

FIGURE 16 Device Input Thread

6.2 Data Objects

The Device Input Thread will have priority access to the Device/Event object category, but will primarily read these objects. Note that queued events will be read and deleted by the CIT after a playback request is made, or when the device is thawed. Access to the event related objects will be accessed through the Device/Event Object Monitor. See CHAPTER 11 for a list of the event related objects.

6.3 Concurrency Issues

The DIT will place an element on the Pending Operation Queue (POQ) and set the appropriate conflict mask bits. The DIT must contend with other CIT threads for access to the POQ.

The DIT also contends with CITs for access to the Device/Event Object when propagating the event up the window tree (see CHAPTER 11).

6.4 Internal Organization and Implementation

6.4.1 General Thread of Control

In the MST:

- setup a socketpair for the DIT using two fake file descriptors
- add the second fake file descriptor to the DIT's select mask to notify select that some device or the DIT has changed state

```
DIT()
{
        setup dummy client record for the DIT
        register a local message buffer for the DIT
        unblock SIGALRM signal

        insert POQ entry
        set screen saver
        remove POQ entry

        while(1)
        {
                update select mask
                select
                if (select fails)
                        if (interrupted by signal)
                                continue
                        else
                                break
                if fake file descriptor has been selected
                        read value from socket
                        switch on value
                                device_enabled_or_disabled:
                                        continue
                                set_screen_saver:
                                        insert POQ entry
                                        set screen saver
                                        remove POQ entry
                                destroy_DIT:
                                        break

                insert POQ entry
                process input events, propagate, and build event list
                set screen saver
                remove POQ entry
                FlushAllMessages()
        }

        unregister a local message buffer for the DIT
        free dummy client
```

```
        lock DITMutex;
        signal DITCond
        unlock DITMutex;
        pthread_exit();
}
```

The DIT works on one device event at a time. Each device event is propagated from the event window up the window tree to the root. See CHAPTER 11 for details about this process.

6.4.2 Communication with the DIT using a socketpair

When the DIT is initialized by the MST, two socketpair file descriptors are initialized. The purpose of the socketpair is to allow other threads to notify the DIT when device state changes, or the DIT should exit. Since the DIT could be waiting on the *select*() to finish, adding the second socket of the socketpair to the select mask insures that the DIT is notified as soon as possible.

When a thread writes a value into the first socket of the socketpair, the DIT reads that value from the second socket of the socketpair and takes the actions as outlined above.

The value *device_enabled_or_disabled* is written into the first file descriptor by the *AddEnabledDevices*() and the *RemoveEnabledDevices*() functions when the active device database changes. This mechanism is also used to notify the DIT that the screen saver should be reset, and is used by the MST to destroy the DIT during server cleanup.

6.4.3 Client Event Delivery

The DIT will receive raw device events, locate the event window, and propagate the events up the tree. As the DIT builds this list of events, it will insert them in its local message buffers. When all events are built, the DIT will create a COT(s) to deliver the events, if deliverable, to the correct client socket.

6.4.4 Event Processing

The DIT must take in raw device input, both from the core devices and any extension devices. In the case of the core devices, KeyPress and KeyRelease are accepted from the Keyboard device, and ButtonPress, ButtonRelease, and MotionNotify are accepted from the Mouse device. Each raw event is processed as described in CHAPTER 11.

6.4.5 DDX Software Cursor

See Section 14.1 in CHAPTER 14.

6.4.6 ScreenSaver

If ScreenSaver is OFF and the DIT detects that no input has been received during the ScreenSaverInterval, the DIT will set ScreenSaver ON.

If ScreenSaver is set ON and device input is received, the DIT sets ScreenSaver OFF.

See CHAPTER 15.

6.4.7 R5 versus MTX

The MTX implementation of the DIT *select*s on those devices that were initialized and started by the Main Server Thread when the Device Input Thread was created. The *select* is similar to that found in the *WaitForSomething* module of the R5 implementation.

Another implementation dependency is in the call to process the device input. For instance, on the Omron Luna88k, when a device has input for the selected DIT, the DIT calls *ProcessInputEvents*. These are similar to the calls seen in R5.

The DIT must do blocking I/O, suggested in the structure chart by the *select*. When the thread wakes up it will call *ProcessInputEvents*. The only other changes necessary are pthread locks/unlocks of the DeviceEventMutex before calling any DIX event-related functions (e.g. *processInputProc(), AddInput(), etc.).

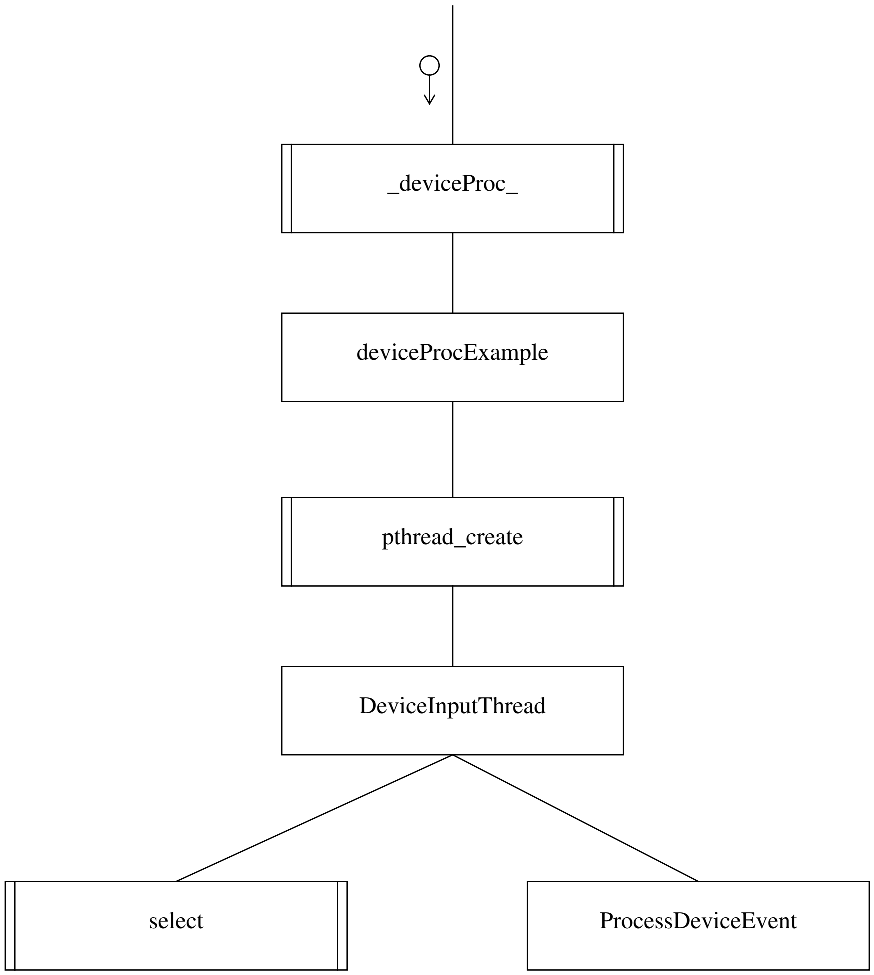

FIGURE 17 DIT deviceProc example Structure Chart

CHAPTER 7

Signal Handling Thread

7.1 Overview

The Signal Handling Thread (SHT) is responsible for detecting and taking action on all process-wide signals that are handled by the MTX server. Assuring that all server processed signals are delivered to the SHT will relieve all other threads from the responsibility of providing for potential interrupts during certain system calls.

7.2 Data Objects

A signal set consisting of all signals to be processed by the MTX server is initialized at server start-up time and then accessed but never modified. The only data modified by the SHT is the global flag, serverException, which indicates whether the server should be terminated or recycled.

7.3 Concurrency Issues

The SHT is created by the MST during initial start-up and remains active until the server terminates.

The set of all signals to be processed by the server must be initialized and blocked by the MST prior to any other thread being created. When new threads are eventually created, they will inherit the set of blocked signals as created by the MST. The SHT will specifically ask for delivery of blocked signals, thus assuring all server processed signals are delivered only to the SHT.

7.4 Internal Organization and Implementation

7.4.1 Processed Signals

The MTX server will catch the following signals and take the associated action.

- **SIGHUP:** Signal the MST to recycle the server.
- **SIGINT:** Signal the MST to terminate the server.
- **SIGTERM:** Signal the MST to terminate the server.
- **SIGALRM:** Signal the DIT to enable screen saver.
- **SIGPIPE:** Catches and ignores signal.

7.4.2 SHT Algorithm

The MST is the only thread that modifies its signal state and does this prior to creating any new threads. Threads inherit the signal state of their creator, thus, all threads will have identical signal states since the MST is at the root of the thread hierarchy (note current exception to this in Section 7.4.3). Prior to creating any threads, the MST will block all signals that are to be processed by the MTX server. A signal that is blocked by all threads will pend until the signal state of the threads changes (never for MTX) or a thread requests delivery of a pending signal via the sigwait() system call. The following is the SHT algorithm.

```
SHT()
{
        for (;;)
            switch (sigwait(<mtx signal mask>)
            {
                case SIGINT:
                case SIGTERM:
                    signal MST to terminate server
                    break
                case SIGHUP:
                    signal MST to reset server
                    break
                case SIGALRM:
                    enable screen saver
                    break
                case SIGPIPE:
                default:
                    break
            }
}
```

7.4.3 SIGALRM Delivery

SIGALRM is special cased in the implementation as of this writing. It is unclear in Draft 6 of P1003.4a as to whether SIGALRM is delivered like other signals on a process-wide basis, or if it is delivered only to the thread that initiates the alarm. The current implementation allows for SIGALRM to be delivered only to the thread that initiates the alarm. Only the DIT will initiate an alarm and thus must be capable of receiving a SIGALRM. If a future Draft of P1003.4a defines SIGALRM delivery to be process-wide, the implementation could be cleaned up by assuming all signals are handled via the SHT.