Version 3

Scalable Message Oriented Middleware for Distributed Computing

MomSys

User Guide

Printed in United States of America

X•IPC VERSION 3.3.0

MomSys USER GUIDE

Table of Contents

1. INTRODUCTION

X•IPC Version 3, a Message Oriented Middleware product from Envoy Technologies Inc., is arguably the most advanced application messaging middleware product in the industry. *X•IPC* Version 3 defines a new level of Message Oriented Middleware technology.

The primary goal of *X•IPC* Version 3 has been to provide features which address the new generation of large-scale distributed and client/server applications. To that end, numerous capabilities have been incorporated in the product for building large, highly-scalable, enterprise messaging applications.

X•IPC Version 3 is a multi-modal communications toolset that is comprised of subsystems for supporting a variety of communication modes. Included are mechanisms for program-to-program *messaging*, *memory-sharing* and *semaphore-synchronization*.

The essential focus of *X•IPC* Version 3 is application-to-application message communication (i.e., application messaging). This class of technology is known in the industry as *Message Oriented Middleware*. A new subsystem, the Message Oriented Middleware Subsystem, or "*MomSys,*" is introduced in *X•IPC* Version 3 as the focal point of the new release. With *X•IPC* MomSys, it is possible to address a wide cross-section of messaging application requirements, ranging from:

Small LAN-based applications, to large WAN Enterprise or Internet applications;

Synchronous, on-line, client/server applications, to asynchronous, disconnected mobile applications;

High-performance, memory-based messaging applications, to industrially-strong, store-and-forward, guaranteed message delivery applications.

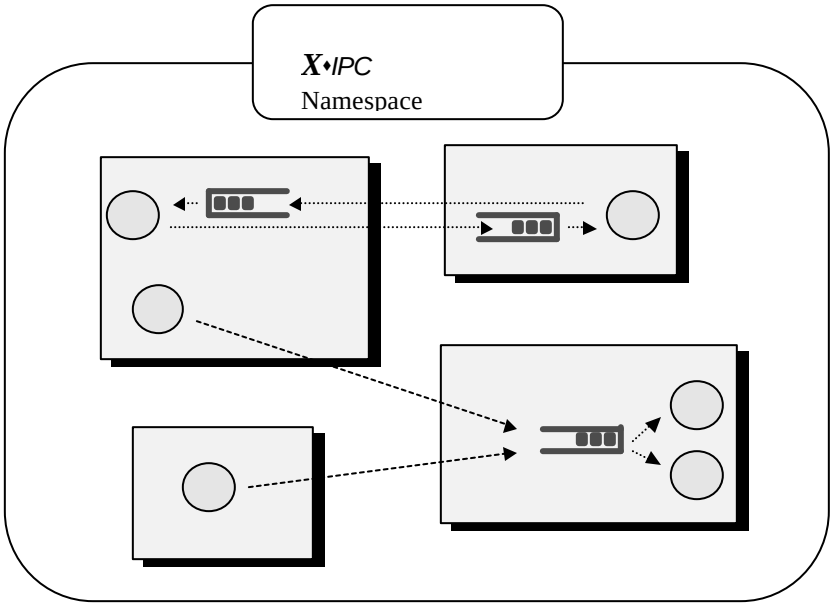
This document describes the *X•IPC* Version 3 MomSys subsystem, its programming model, functionality and application programming interface (API). It outlines as well some of the more notable messaging features inherent in the product for building, configuring and scaling the full range of messaging applications.

2. MOMSYS ARCHITECTURE AND PROGRAMMING MODEL

This section presents a top-down sketch of the architecture and programming components of *X♦IPC* Version 3's MomSys subsystem.

2.1 The "30,000 Foot" View

At a very high-level, MomSys has the following appearance:



The model depicted is rather simple: Application programs (circles in the above diagram), send messages to other programs by placing them onto "*Application Queues*" (open-rectangles in the above diagram) from which they are read by the targeted programs.

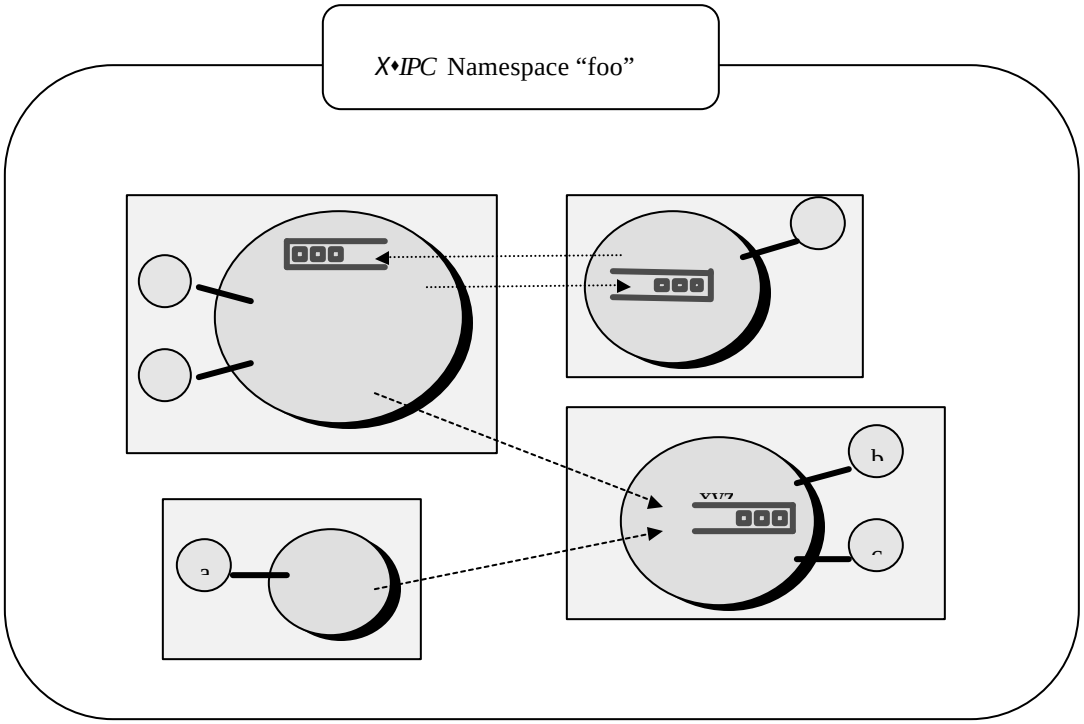
Message movement between nodes follow a store-and-forward route (dashed-lines above) for providing guaranteed and possibly deferred message delivery.

2.1.1 X♦IPC NAMESPACE

Application queue names and locations are managed globally within an *X♦IPC* namespace. An *X♦IPC* namespace is implemented via fault-tolerant namespace catalog servers. A program need *not* know the location of an application queue when sending it messages. It need only reference a targeted app-queue by its *name*. Mapping between app-queue name and app-queue location is handled dynamically by *X♦IPC*.

2.2 The “20,000 Foot” View

A closer look at the programming model reveals the following additional detail:



2.2.1 “LOCAL INSTANCES”

A process wishing to perform work within an X•IPC namespace must first log into an X•IPC instance that is affiliated with the namespace. This instance, which is typically local to the process (i.e., on the same network node), acts as an access point into the X•IPC namespace. (For an introductory discussion on the topic of X•IPC Instances, refer to “X•IPC Instances” in the X•IPC User Guide.)

Consider the above diagram. Each network node (*rectangle*) contains an X•IPC instance (*shaded ovals*).These instances act as access points for processes wishing to perform messaging operations within the depicted X•IPC namespace named “foo” (*the large rounded rectangle*). Processes (*circles*) log into local instances for accessing the X•IPC namespace. Once a process logs into to an instance it gains access to the X•IPC namespace with which that instance is affiliated. That instance is referred to as the process’ “local instance” within the namespace.

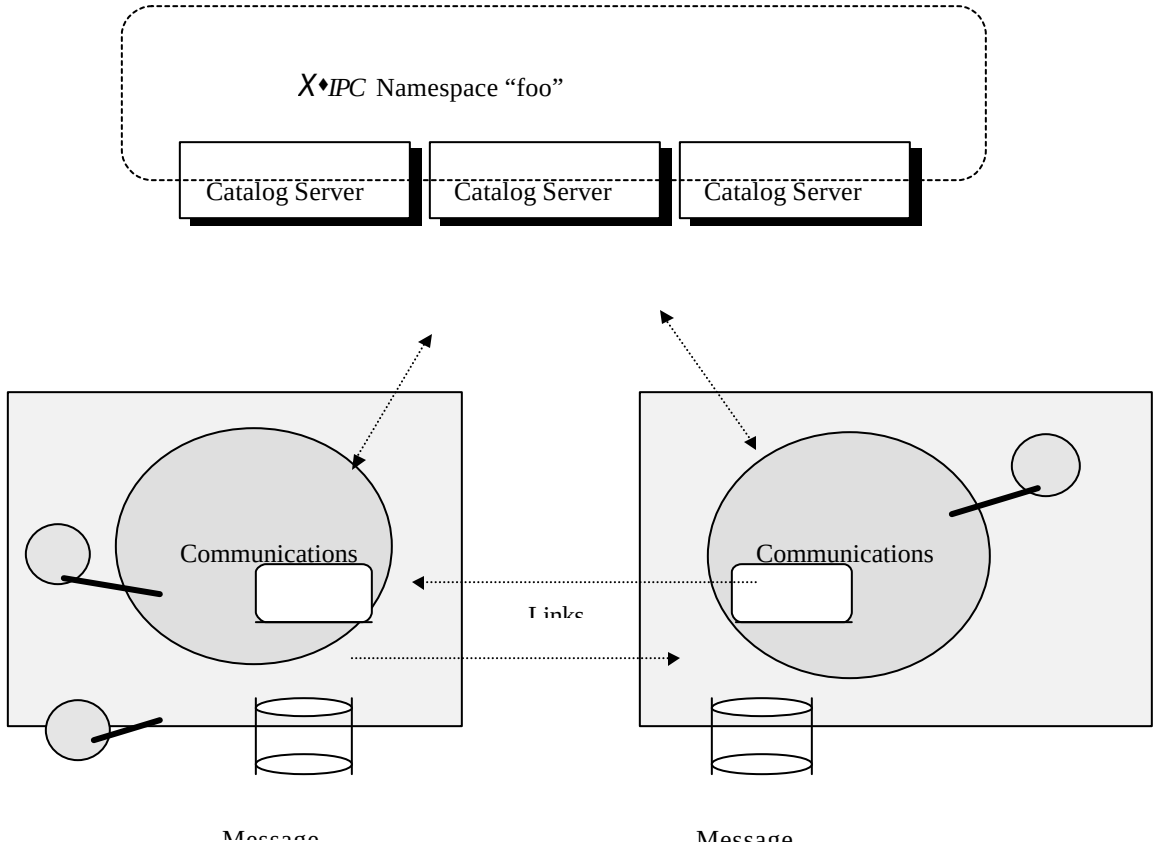
(An instance has an affiliation with a single X•IPC namespace. For situations where a process needs to work with a second namespace, a second local instance, affiliated with the second namespace, may be accessed and the process can then toggle between the two, as needed. This is an advanced topic that will be addressed later in this guide.)

Processes which plan to receive messages create application-queues (abbreviated as “*app-queues*”) that are physically located within their local instance. The names of created app-queues may be known only within the confines of the local instance, in which case they are only accessible by other process logged into the local instance, or they may be registered within an X•IPC namespace, so that they can be reached by processes spread over the network.

A process (such as “a” in the above diagram) sending messages to an app-queue (“xyz”) targets the app-queue by specifying its name. X♦IPC transmits the messages to the targeted app-queue, wherever it is within the namespace. Enqueued messages are subsequently received from the app-queue by processes local to the app-queue (“b” and “c”).

2.3 The “10,000 Foot” View

Working down to a level of still greater detail, we see the following:



2.3.1 X♦IPC CATALOG SERVERS

X♦IPC namespaces are implemented within a set of (one or more) redundant X♦IPC catalog servers. We will see that these programs support the network-transparent app-queue discovery mechanism within MomSys.

Namespace information may be automatically replicated between the catalog servers for two possible purposes:

Fault Tolerance - Should any one of the catalog servers fail, access to the namespace data is unaffected. This automatic fail-over is dynamic, and transparent to the user.

Locality - If a namespace is to transcend a wide geographic region, it is possible to strategically position multiple catalog servers for providing localized access to the single namespace via its replicated namespace data. Here again, if one of the local catalog servers were to fail, the affected users would automatically start accessing one of the other available servers.

2.3.2 MESSAGE REPOSITORY

Within each instance is a non-volatile *message repository* (indicated by the disk-drive, in the above diagram) This is used by X♦IPC MomSys for storing and tracking message movement between instances as they travel from sender

to receiver in a store-and-forward, asynchronous, guaranteed delivery manner. Messages being sent to a disk-based app-queue are moved via the local and then remote message repositories. This guarantees that messages are not lost as they go from sender to receiver.

A comprehensive set of message tracking-levels is provided for directing X•IPC, per message, as to just how far along the message path tracking is desired. Event-driven message tracking is provided as well.

A critical component of a store and forward, message-oriented middleware technology that is to be used in "real-world" applications is its ability to provide message tracking tools for accessing immediate and up-to-date status information about previously sent messages. It is imperative that the user be allowed to immediately ascertain where any previously sent message currently resides. X•IPC MomSys provides facilities for supporting this function.

Similarly, X•IPC provides a full set of options for defining message expiration periods, message repository clean-up scheduling and message journaling.

2.3.3 COMMUNICATION MANAGER

2.3.3.1 Concept

Within each instance is a multi-threaded *communication manager* (indicated by the rounded rectangles, in the above diagram). A communication manager supports an instance's affiliation with an X•IPC namespace as well as its communication links with other instances.

The communication manager is responsible to take outgoing messages from the local message repository and send them to a counterpart communication manager in a destination instance. The destination communication manager stores the messages in its message repository. The degree of internal message acknowledgments that occurs between communication managers depends on the message tracking level specified per each message sent. (Message tracking levels are described in detail in Appendix A "Message Status Values and Tracking Levels".)

The communication manager additionally maintains a local cache of app-queue names and locations. It uses this data to route outbound messages. Accordingly, it obtains periodic data updates from the X•IPC catalog with namespace updates and related data. These updates *do not* occur in a broadcast manner, but are rather designed to take place only as necessary.

2.3.3.2 Design

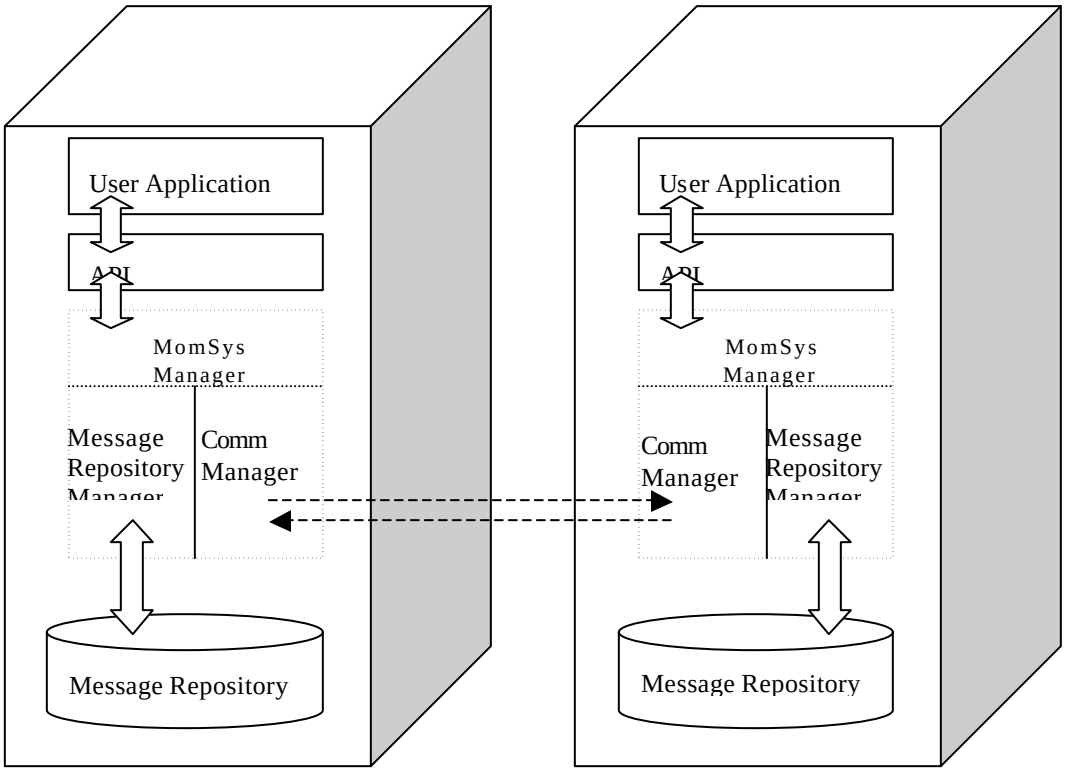
The communication manager is implemented as a set of process-pairs called communication servers. The communication server programs control simplex communication lines - outgoing and incoming - with remote instances. (The term "process" is used in its descriptive sense; operating system processes and "threads" are used for actual implementation). This design allows X•IPC to keep multiple messages in flight over a network link at any point in time. This asynchronous communication takes full advantage of the underlying network protocol bandwidth.

Each communication manager handles multiple sessions, sending or receiving messages from multiple instances. This architecture enables large-scale client/server implementations to be handled without consuming inordinate amounts of computer resources.

The communication manager is configurable with regard to the number of concurrent sessions to keep, timeout periods, retry intervals, catalog search intervals, and so on. These parameters are defined in the MomSys Reference Manual, and will be listed at a later point in this User Guide.

2.4 The Reliable Messaging Programming Model

The *X•IPC* reliable messaging model is manifest in the MomSys subsystem of *X•IPC* . It comprises an API, a Subsystem Manager, a Message Repository Manager and a scalable Communication Manager working as a unit, and communicating with the MomSys subsystems of the other remote instance. The basic MomSys architecture is depicted below.

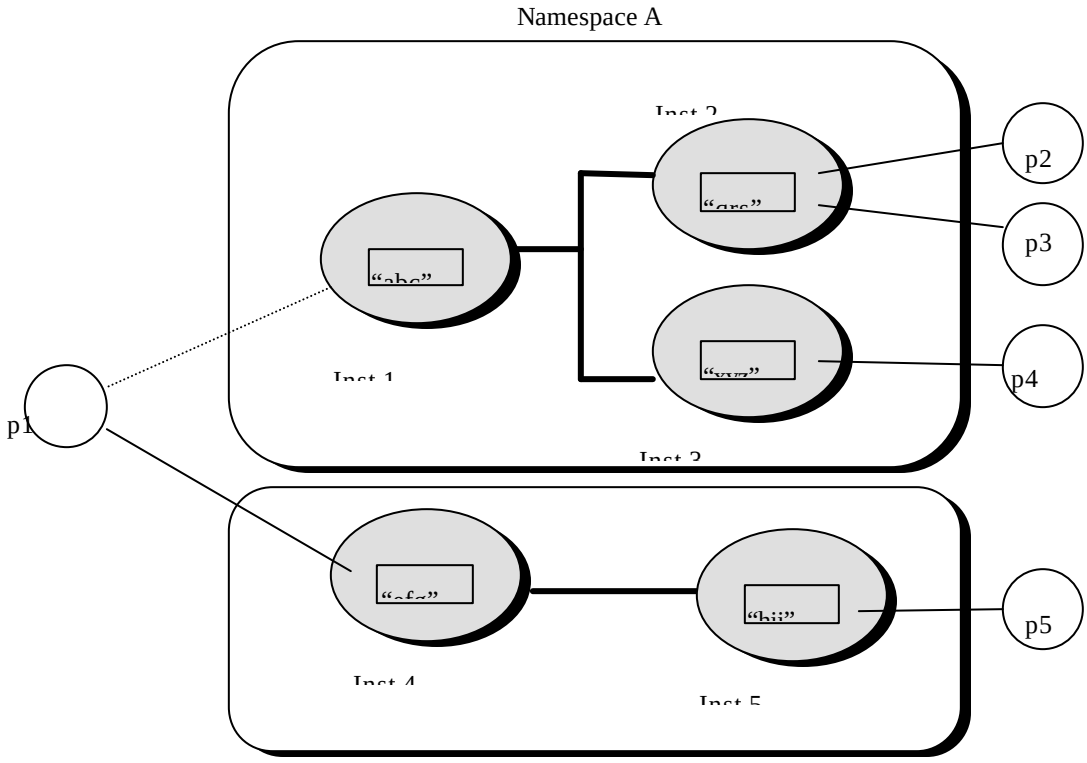


The MomSys Programming Model

Copies of sent messages are stored locally until they are known to have been successfully delivered.

2.5 MomSys Terminology

The following is a brief outline of the MomSys programming model components and the definitions that emerge. Consider the following diagram:



We see two X•IPC namespaces, A and B. Within namespace A are three instances. Three app-queues named “abc”, “qrs” and “xyz” are situated in the three instances. We similarly see two instances each containing an app-queue, within namespace B. (An instance, of course, can support many app-queues. The above example has one app-queue per instance for purposes of clarity.) The solid and dashed lines indicate user processes logged into X•IPC instances. Solid lines are connected X•IPC logins; dashed lines are disconnected X•IPC logins.

We also see processes p1 through p5 that have logged into the X•IPC instances as indicated. The dashed line between process p1 and Inst-1 indicates that the process has disconnected from that login. Its current login is to Inst-4. For a complete discussion of working with multiple X•IPC instances refer to “Working With X•IPC Instances” in the Advanced Topics section of the X•IPC User Guide.

The following definitions clarify the terms and their relationships:

X•IPC Namespace

An X•IPC namespace is a collection of X•IPC named entities. In MomSys these entities are typically named application queues. As shown earlier, a namespace is managed by a set of catalog servers.

An Instance’s Affiliated Namespace

An X•IPC instance may be affiliated with at most one namespace. This is referred to as the instance’s affiliated namespace.

Local Instance

A process must log into an X•IPC instance that is affiliated with a namespace before performing MomSys operations within that namespace. This instance is referred to as the process’ local instance within that namespace. A process wishing to work additionally within a second namespace may do so by logging into an instance within that second namespace. Such a process is said to have two local instances, one per namespace.

Current Local Instance, Current Namespace

The instance that a process is currently connected to is the process' *Current Local Instance*. The corresponding namespace is the process' *Current Namespace.*

In the preceding diagram, Process p1 is working within namespaces A and B, and has accordingly logged into Inst-1 and Inst-4. These are p1's local instances. Note that p1's login to Inst-4 is its current login. Hence, Inst-4 is p1's current local instance and namespace B is its current namespace.

Similarly, p2, p3 and p4 are working in namespace A. Both p2 and p3 are using Inst-2 as their local instance. Process p4 is using Inst-3. Process p5, however, is working within namespace B. Its local instance is Inst-5.

Remote Instance

As just described, when a process works within a namespace, the instance that it logs into within that namespace is that process' local instance within that namespace. Remaining instances within that namespace, i.e., those *not* logged into, are *remote instances*, relative to it. For example, Inst-2 and Inst-3 are p1's remote instances within namespace A. Inst-5 is a remote instance, relative to p1, within namespace B.

Local App-Queue

App-queues situated within a process' local instance are referred to as local app-queues. For example, app-queue "abc" in the above diagram is local relative to process p1. Similarly, app-queue "qrs" is local relative to processes p2 and p3.

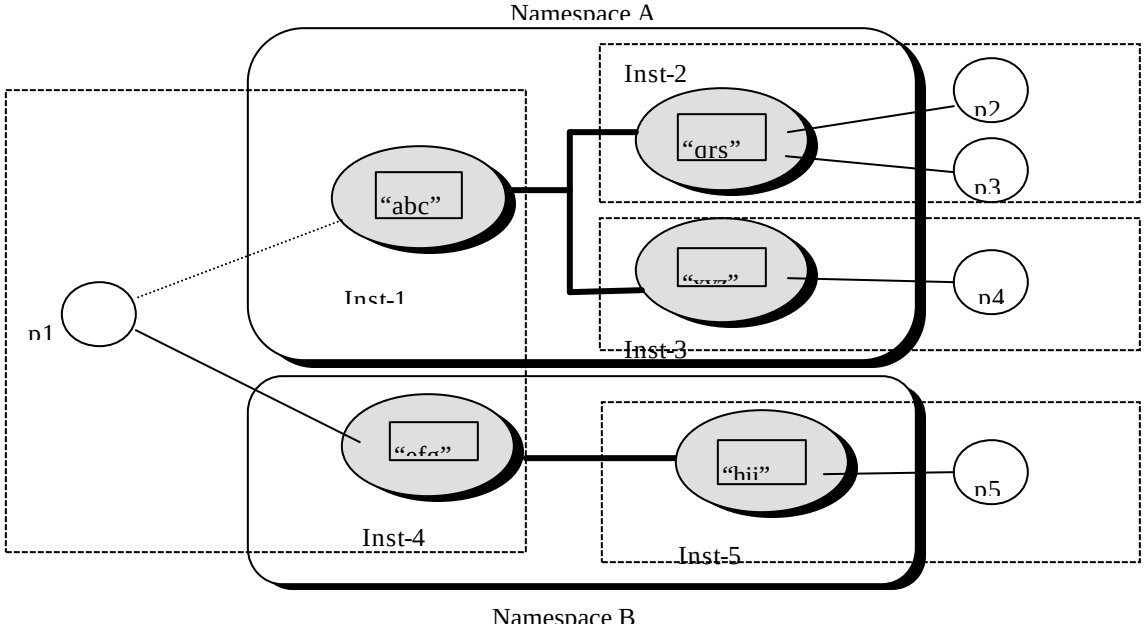
A process may perform all forms of app-queue manipulation operations on local app-queues, such as: MomSend(), MomReceive(), MomDelete(), MomDestroy(), MomInfoAppQueue(), etc.

Remote App-Queue

App-queues situated within a process' remote instances are referred to as remote app-queues. For example, app-queue "qrs" is a remote app-queue relative to process p1. Similarly, app-queue "abc" is a remote app-queue relative to processes p2 and p3.

The only operations which may be performed on remote app-queues are MomAccess(), MomSend() and MomInfoAppQueue().

The above environment may deployed in a variety of scenarios without affecting the overall model. One possibility involving four network nodes (depicted as dashed-line boxes) is the following:



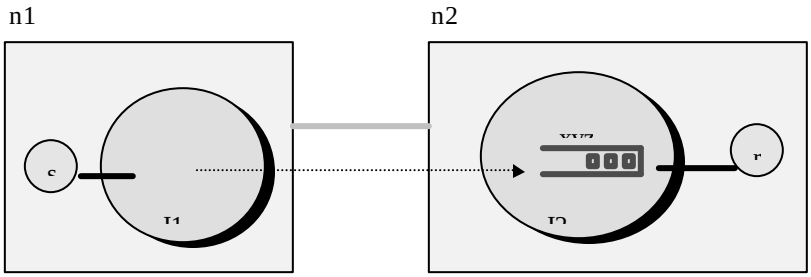
Note that one of the platforms supports more than one *X•IPC* instance. Specifically, Inst-1 and Inst-4 both reside on a single platform. Process p1 employs them for accessing the two namespaces A and B.

Obviously, other scenarios are possible without changing any aspects of *X•IPC* 's utilization.

3. BUILDING A SIMPLE MOMSYS APPLICATION

In this section we will program a very simple MomSys application in which a program will send *one* guaranteed delivery message to a second program. The point of this exercise is to demonstrate, by example, the MomSys programming model. Following this example, it should be possible to start "connecting the dots" as to how MomSys works, and how it will be useful for building mission-critical messaging applications.

3.1 The Environment



Our environment for this exercise will assume two nodes on a TCP/IP network having names n1 and n2. We will further assume that the sending program "s" is running on n1 and will employ a local *X•IPC* instance I1; and that the receiving program "r" is running on n2 and will employ local *X•IPC* instance I2.

The focus of our application is app-queue "xyz," a disk-based app-queue created within instance I2 by receiver program "r". Sender program "s" will send its message to app-queue "xyz". Receiver program "r' will receive the message. And with that, the application will end. (Note, of course, that the two programs may run in any sequence.)

3.2 Programming Steps

3.2.1 SENDER PSEUDO-CODE

Program "s" will be coded to perform the following steps:

Log into instance I1.

Access a handle for app-queue "xyz".

Send message to app-queue "xyz".

Log out of instance I1.

3.2.2 RECEIVER PSEUDO-CODE

Program "r" will be coded to perform the following steps:

Log into instance I2.

Create app-queue "xyz".

Receive message from "xyz".

Log out of instance I2.

3.2.3 SENDER PROGRAM

The following is the program for "s". Note: the datatypes VOID, XINT and CHAR are X•IPC provided datatypes for enhancing an application's portability between disparate machine architectures.

```
/*
 * Sender program "s". Note, error checking is omitted
 * for enhancing program readability.
 */

#include "xipc.h"

VOID
main( argc, argv )
XINT argc;
CHAR **argv;

{
    XINT xyzAQid;

    XipcLogin("@I1", "s");

    /*
     * MomAccess() accesses a handle to app-queue "xyz".
     * The NOVERIFY flag specifies to XIPC that
     * MomAccess should not verify whether the app-
     * queue already exists. In a case where
     * the app-queue is not yet known, MomAccess will
     * return a "virtual" AQid handle. This is invisible
     * to the program.
     */

    xyzAQid = MomAccess(MOM_NOVERIFY("@xyz"));

    MomSend(
            xyzAQid,                /* The AQid handle of app-queue "xyz" */
            "Hello world",          /* Message being sent */
            12L,                    /* Size (in bytes) of message */
            MOM_PRIORITY_NORMAL,    /* Priority of sent message */
            MOM_TRACK_DELIVERED,    /* Track message until it is delivered */
            MOM_REPLY_NONE,         /* No response expected */
            NULL,                   /* No need to know message-id */
            MOM_WAIT                /* Block if system is busy */
          );

    XipcLogout();
}
```

3.2.4 RECEIVER PROGRAM

The following is the program for “r”. Note: the datatypes `VOID`, `XINT` and `CHAR` are *X•IPC* provided datatypes for enhancing an application’s portability between disparate machine architectures.

```
/*
 * Receiver program “r”. Note, error checking is omitted
 * for enhancing program readability.
 */

#include “xipc.h”

VOID
main( argc, argv )
XINT argc;
CHAR **argv;

{
    XINT xyzAQid;
    XINT InBufferLen = 12;
    CHAR InBuffer[12];

    XipcLogin(“@I2”, “r”);

    xyzAQid = MomCreate(“xyz”, MOM_APPQUEUE_DISK_REGISTER);

    MomReceive(
            xyzAQid,                /* Handle of app-queue “xyz” */
            InBuffer                /* Where to read message */
            InBufferLen,            /* Size (in bytes) of buffer */
            MOM_MESSAGE_FIRST,      /* Get first msg from app-queue */
            NULL,                   /* (we don’t need ReplyAQid) */
            NULL,                   /* (we don’t need MsgId) */
            NULL,                   /* (we don’t need detailed Msg Info ) */
            MOM_WAIT                /* Block message isn’t there */
           );

    printf(“got message: %s\n”, InBuffer);

    XipcLogout();
}
```

3.3 Summary

Upon reviewing the above programs its should be evident what is occurring. There are, however, a few important points, that may not be obvious, and that are worthy of mention:

It does *not* matter which program is started first. If the sender program “s” runs before the receiver “r” has started, the sent message will be held within the message repository of instance I1 until “r” has started and created app-queue “xyz”. This may occur a second later, a minute later or a week later.

In fact, node n2 may be shut down and off the network entirely at the time that “s” runs. It has no effect on sender program “s”.

The sender program may “fire and forget”. It completes as soon as the message has been submitted within its local instance. Once the message is sent, it is *X•IPC* ‘s responsibility to move the message forward as fast as possible, independent of the sender.

The sent message may be tracked at any point in time to determine its latest status.

4. BASIC MOMSYS PROGRAMMING FUNCTIONALITY

4.1 Creating an App-Queue

4.1.1 WHAT IS AN APP-QUEUE?

Before addressing the topic of app-queue manipulation, it is instructive to first understand what an app-queue is. *An app-queue is a set of messages that are maintained according to a certain logical sequence.* This sequence is known as the app-queue's "natural" sequence.

4.1.1.1 Natural Sequence

Every app-queue that is created has, as one of its defining attributes, a natural sequencing of its messages. This is referred to as the app-queue's natural message sequence. There are two possible natural sequences:

Time sequence

Priority sequence

By default, an app-queue's natural sequence is the *time* sequence in which the messages arrive and are placed on the queue, i.e., FIFO sequence. We will see later that the MomAttrSet() function can be used to override this default to create an app-queue whose messages are sequenced by *priority*, i.e., highest priority at the front of the app-queue.

An app-queue's natural sequence defines the order in which messages are presented to users performing MomReceive() operations on that app-queue. The topic of message selection will be addressed later, in the description of the MomReceive() function call.

4.1.2 BASIC APP-QUEUE ATTRIBUTES

An app-queue is defined by other attributes in addition to those that define its message sequencing. The following are brief descriptions of the basic app-queue attributes. Advanced app-queue attributes are described in a later section. The complete list of app-queue attributes, their descriptions and default settings are defined in the MomAttrSet() function definition in the MomSys Reference Manual.

4.1.2.1 The Time Sequence Attribute

The `MOM_ATTR_SET_TIME` app-queue attribute indicates that the app-queue employs a natural message sequencing that stores incoming messages in *time* order (i.e., oldest arriving message at the front).

4.1.2.2 The Priority Sequence Attribute

The `MOM_ATTR_SET_PRIORITY` app-queue attribute indicates that the app-queue employs a natural message sequencing that stores incoming messages in *priority* order (i.e., highest priority message at the front).

The Time and Priority attributes are mutually exclusive.

4.1.2.3 The Disk-Based Communication Attribute

The `MOM_ATTR_SET_DISK` app-queue attribute indicates that the communication of messages to the app-queue occurs via store-and-forward disk-based mechanisms. This can support the asynchronous guaranteed message delivery of sent messages.

4.1.2.4 The Automatic Namespace Registration Attribute

A related attribute, the `MOM_ATTR_SET_AUTO_REGISTER` app-queue attribute, indicates that an app-queue is *automatically* registered and deregistered within an *X•IPC* namespace upon its creation and deletion from its local

instance. This is useful for developing applications that need to automatically insert and delete X•IPC namespace entries without requiring program intervention.

4.1.2.5 The Automatic Namespace Registration Update Attribute

`MOM_ATTR_SET_AUTO_REGISTER_UPDATE` updates an app-queue's registration data. This attribute is typically used to relocate an app-queue from its current location to a new location and to have all programs that are currently sending messages to that app-queue have their messages subsequently be sent to the new location. (See Section 4.1.6 for further details.)

4.1.3 THE MomCreate() FUNCTION

The MomCreate() API accepts two arguments: the name of the app-queue to be created, and a pointer to a data structure of type `MOM_ATTRBLOCK_APPQUEUE`, as follows:

- The name argument is the name (i.e., character string) by which other programs will reference the app-queue, once created. Alternatively, it is possible to create an app-queue having no name. This is called a "private" app-queue, and it is created by specifying MOM_PRIVATE as the name argument to MomCreate(). Such an app-queue is typically used in client/server communication settings, in which each client creates its own private response app-queue, instead of having to invent a unique client-side app-queue name. This is demonstrated later in this Guide in the section "Client/Server Interaction."

- The second argument points to a `MOM_ATTRBLOCK_APPQUEUE` data block that describes the nature of the app-queue to be created. Attribute values within the block may be set "manually" via the MomAttrSet() API, in which case MomAttrSet() is called for setting each of the individual app-queue attributes; or alternatively the programmer may employ one of the predefined attribute blocks provided by X•IPC. These are listed in the next section. Examples of using both approaches are provided below.

MomCreate() will fail if an app-queue with the specified name (other than `MOM_PRIVATE`) already exists within the caller's current X•IPC instance. Similarly, if the app-queue is being registered within an X•IPC namespace, MomCreate() will fail if an app-queue with the specified name already exists within the caller's current namespace.

4.1.4 PREDEFINED MOM_ATTRBLOCK_APPQUEUE BLOCKS

The following predefined app-queue attribute blocks are provided by X•IPC for streamlining the coding necessary for creating app-queues in many situations:

- `MOM_APPQUEUE_DISK` - is used for creating an app-queue that has all default attribute settings. Default attribute settings are listed in the MomAttrSet() manual page definition.

- `MOM_APPQUEUE_DISK_REGISTER` - is used for creating an app-queue that has default attribute settings, with the exception that the created app-queue is automatically registered within the caller's current namespace. (Auto-registration is *not* the default) It is an error to register an app-queue name that is already registered.

- `MOM_APPQUEUE_DISK_REGISTER_UPDATE` - is used for creating an app-queue that has default attribute settings, with the exception that the created app-queue is automatically registered within the caller's current namespace. (Auto-registration is *not* the default) In case the app-queue already exists, its attributes are updated with the attributes passed in the current call. It is typically used to relocate an app-queue from its current location to a new location and to have all programs that are currently sending messages to that app-queue have their messages now be sent to the new location. (See Section 4.1.6 for further details.)

4.1.5 EXAMPLES OF CREATING AN APP-QUEUE

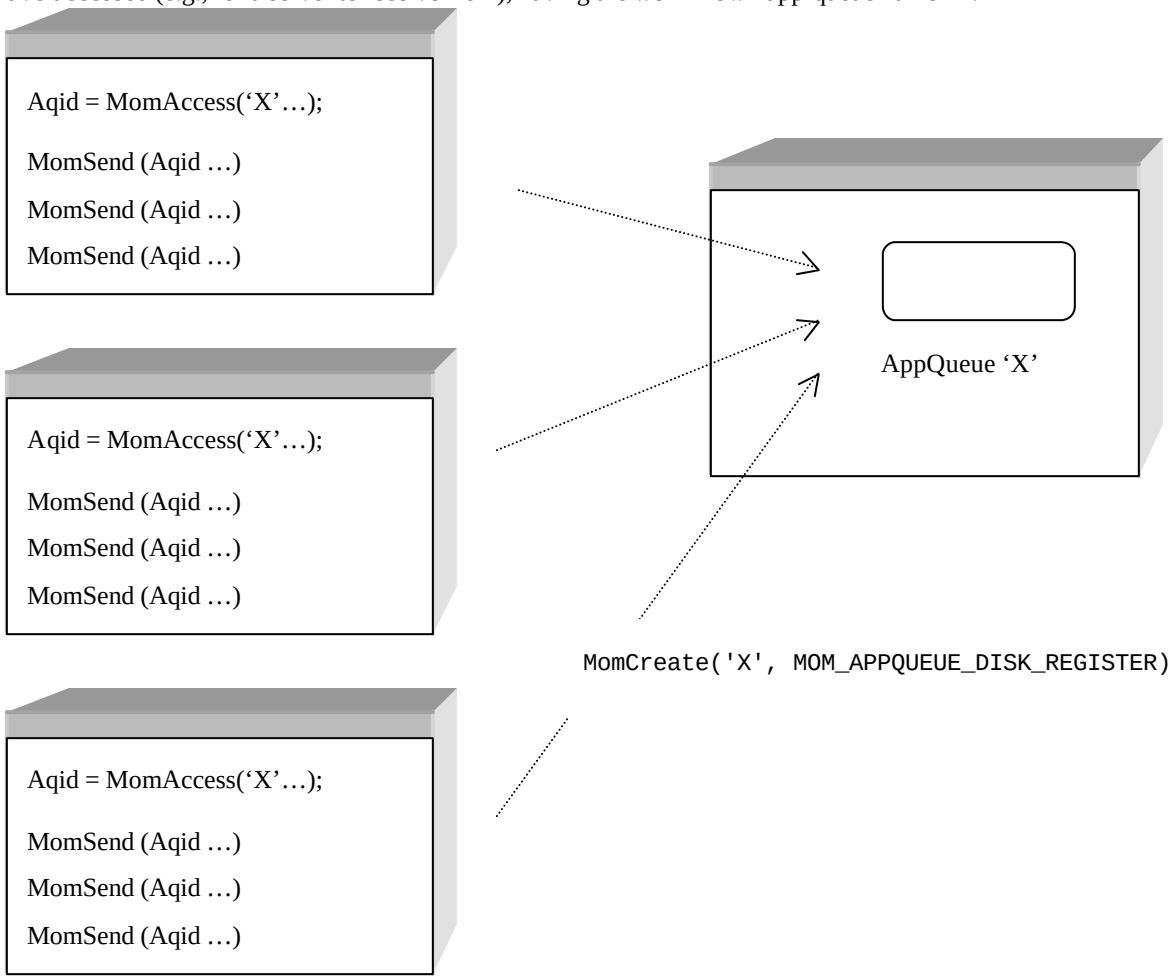
The following sample code segments demonstrate the creation of a variety of app-queues:

```
/*
 * Create an app-queue named “abc” that is not automatically registered
 * in the XIPC namespace. Default attributes are applied to the created
 * app-queue. This means that the “natural” sequencing of messages on the
 * app-queue will be by time of arrival, i.e., FIFO sequencing of messages.
 */

RetCode = MomCreate (“abc”, MOM_APPQUEUE_DISK);

/*
 * Create an app-queue named “def” that is automatically registered
 * in the XIPC namespace. Default attributes are applied to the created
 * app-queue. This means that the “natural” sequencing of messages on the
 * app-queue will be by time of arrival, i.e., FIFO sequencing of messages.
 */

RetCode = MomCreate (“def”, MOM_APPQUEUE_DISK_REGISTER);
/*
 * Create an app-queue named “ghi” that has a priority-based “natural”
 * sequencing of messages, but that is not automatically registered in the
 * caller’s namespace.
 */

MOM_ATTRBLOCK_APPQUEUE   AttrBlock;

RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_INITIALIZE);
RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_DISK);
RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_PRIORITY);

RetCode = MomCreate (“ghi”, &AttrBlock);

/*
 * Create an app-queue named “jkl” that has a priority-based “natural”
 * sequencing of messages, but that is automatically registered in the
 * caller’s namespace.
 */

MOM_ATTRBLOCK_APPQUEUE   AttrBlock;

RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_INITIALIZE);
RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_DISK);
RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_PRIORITY);
RetCode = MomAttrSet( &AttrBlock, MOM_ATTR_SET_AUTO_REGISTER);

RetCode = MomCreate (“jkl”, &AttrBlock);
```

4.1.6 RELOCATING APP-QUEUES

It is possible to relocate an app-queue from its current location to a new location and to have all programs that are currently sending messages to that app-queue have their messages now be sent to the app-queue's new location. This occurs "on-the-fly," without the sending programs needing to make any rerouting provisions in their code and without them being aware of the targeted app-queue's new location.

Consider the following example in which three programs (e.g., clients) are sending messages to an app-queue that they have accessed (e.g., for a server to receive from), having the well-known app-queue name 'X.'

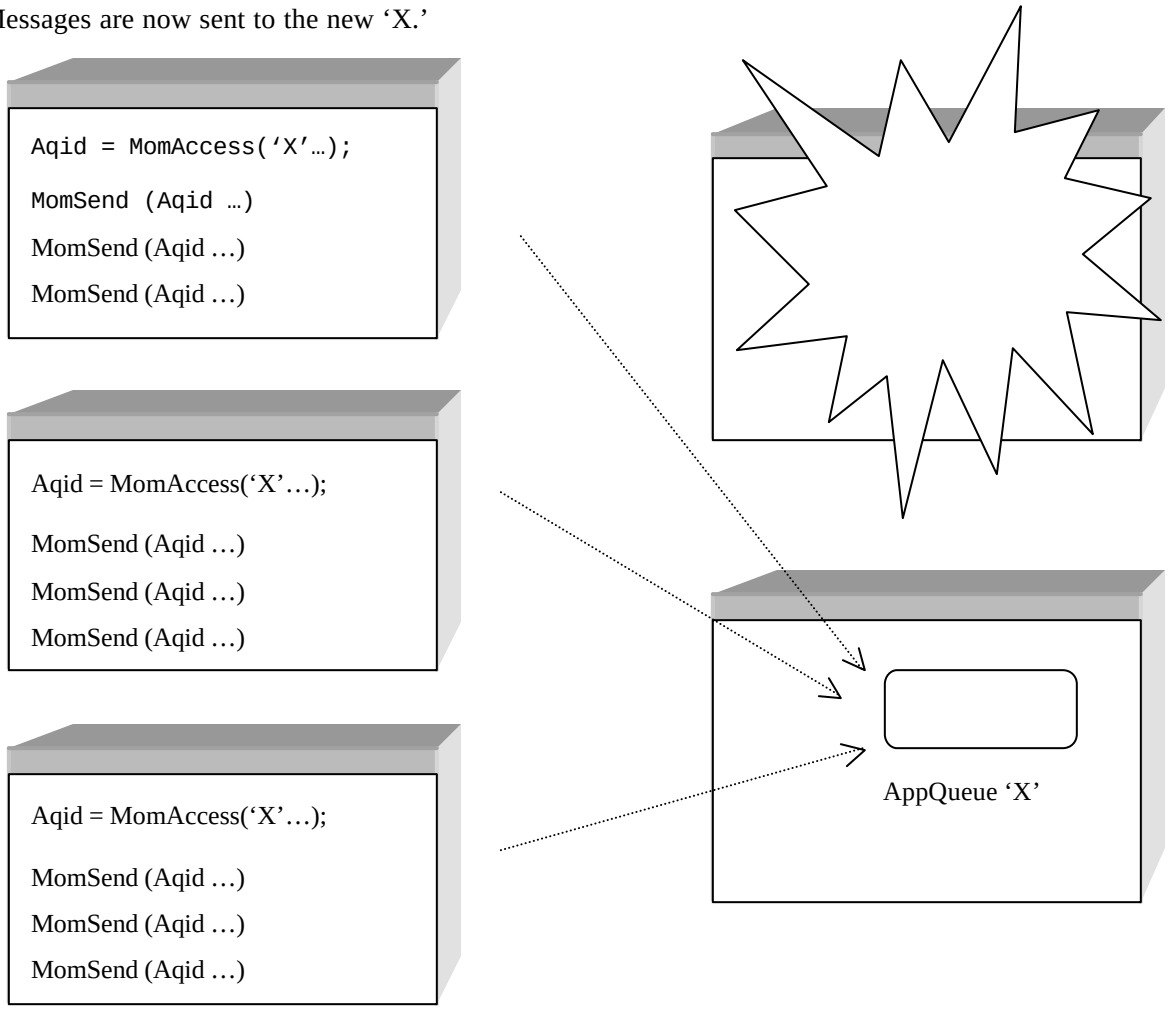


Now, consider what will happen if the server upon which 'X' is located crashes, *X•IPC* allows the user to create a new app-queue 'X' on a new server and, with that, *X•IPC* will cause all subsequent messages sent to 'X' to route to the new 'X.' The client programs themselves remain entirely unaware of this relocation of the app-queue.

By creating the second 'X' and specifying that its registration should *update* the prior registration of 'X,' the new app-queue becomes the target for all messages sent to 'X'.

Even though the sending applications do *not* perform a new `MomAccess('X', ..)` call, the information about the new location of 'X' is automatically disemminated to those nodes on the network having registered a prior interest in such an app-queue.

Messages are now sent to the new ‘X.’




```
MomCreate('X', MOM_APPQUEUE_DISK_REGISTER_UPDATE);
```

4.2 Accessing an App-Queue - MomAccess()

The MomAccess() function provides an AQid (i.e., App-Queue ID) handle to an application program wishing to send messages to an app-queue “somewhere” on the network. MomAccess() takes a single string argument that identifies the name of the targeted app-queue and returns the corresponding AQid..

The *Name* argument to MomAccess() may be specified in a variety of formats for designating a target app-queue. In the following sections we will review some of the possibilities.

4.2.1 LOCAL APP-QUEUE NAMES

A local app-queue is specified to MomAccess() using the string originally passed to the MomCreate() function when the app-queue was created within the calling process' local instance. Thus, the AQid handle of an app-queue that was created locally via MomCreate("abc", …) is accessed via:

```
/*
 * Access an app-queue that is within the caller's current instance.
 */

AQid = MomAccess("abc");
```

A call to MomAccess() specifying a local app-queue will fail if the specified app-queue does not exist within the caller's local instance at the time of the call.

4.2.2 REMOTE APP-QUEUE NAMES

A remote app-queue is specified to MomAccess() using a string starting with the '@' character. A number of variations are possible:

```
/*
 * Access an app-queue that is within the caller's current namespace,
 * having the name "foo".
 */

AQid = MomAccess("@foo");
```

The `"@foo"` argument indicates that the app-queue name "foo" is located somewhere within the calling process' current namespace. When such a name is passed to MomAccess(), X•IPC resolves the location of the app-queue via the X•IPC namespace catalog. This allows processes to send messages to app-queues without knowing where the app-queues are located within the namespace.

```
/*
 * Access an app-queue by means of its fully-qualified identity: network node name;
 * instance name within that node; app-queue name within that instance.
 */

AQid = MomAccess("@SomeNode:SomeInstance:foo");
```

The above example directs MomAccess() that app-queue "`foo`" is to be found within instance "`SomeInstance`", where that instance is to be found on node "`SomeNode`". Such an app-queue is fully qualified to X•IPC thus avoiding the need to query the X•IPC namespace catalog to resolve its location. This is desirable for applications in which the usage of an X•IPC namespace is not appropriate.

4.2.3 VIRTUAL AQID HANDLES

The normal behavior of MomAccess(), when referencing a remote app-queue, is to fail when the specified remote app-queue is not currently verified to exist at the time of the MomAccess() call. In the case that `"@foo"` was specified this will occur because no app-queue having the name "`foo`" was registered at the time of the MomAccess() call. In the case that `"@SomeNode:SomeInstance:foo"` was specified, this will occur if no app-queue "`foo`" is found within instance "`SomeInstance`" on node "`SomeNode`" at the time of the MomAccess() call.

By specifying the *`Name`* argument to MomAccess() as `MOM_NOVERIFY(`*`Name`*`)`, you can force MomAccess() to succeed even if the named app-queue is not found at the time of the call. In such an event, the AQid handle returned by MomAccess() is a *virtual* app-queue handle. Thus, continuing with our "`foo`" example, messages sent using that virtual handle are stored in the local message repository until an app-queue identified as "`foo`" is

subsequently created (and registered, in the case of "`@foo`"). At that point, *X•IPC* forwards messages to that app-queue.

```
/*
 * Access an app-queue that is now or will be registered within the caller's
 * current namespace, having the name "foo".
 */

AQid = MomAccess(MOM_NOVERIFY("@foo"));


/*
 * Access an app-queue that is now or will be located within instance
 * "SomeInstance" on node "SomeNode", having the name "foo".
 */

AQid = MomAccess(MOM_NOVERIFY("@SomeNode:SomeInstance:foo"));
```

We saw an additional example of this in the simple MomSys application built in Chapter 3.

4.2.4 AQID SEMANTICS

The validity-time of AQid values returned by MomCreate() and/or MomAccess() is defined as the period of time during which the AQid can be used for referencing its intended app-queue. This period depends on whether the AQid references a local app-queue (i.e., it is a *LocalAQid*) or the AQid references a remote app-queue (i.e., it is a *RemoteAQid*).

4.2.4.1 LocalAQid Semantics

An AQid of a local app-queue is valid within an instance so long as the app-queue it was originally derived from, via MomCreate() or MomAccess(), still exists within the local instance. Once the app-queue is deleted from the local instance via MomDelete() or MomDestroy(), the LocalAQid is invalidated.

4.2.4.2 RemoteAQid Semantics

An AQid of a remote app-queue (i.e., a RemoteAQid) is valid within an instance so long as the AQid is still being referenced by one or more users of the instance (i.e., they have performed a MomAccess() call referencing the remote app-queue). Following the logout from the instance by the last such user, *X•IPC* invalidates that RemoteAQid (unless there are messages that have not yet been delivered).

*4.2.4.3 De-accessing an App-Queue – **MomDeaccess()***

The MomDeaccess() function frees the association of the user with a remote app-queue that had been previously accessed. As stated above, MomSys keeps track of the number of local users accessing a remote app-queue. When the count drops to zero, MomSys frees the internal resources that supported the remote access. A user logging out from an instance decrements this count. The MomDeaccess() verb enables a program to de-access an app-queue, independent of logging out of the instance. It is a good practice to de-access remote app-queues once they are no longer needed.

AQid should reference a remote app-queue. If *AQid* represents a local app-queue, MomDeaccess() returns "success." This has no effect, however, on MomSys resources.

Example:

```
/*
 * De-access remote app-queue AQid.
 */
```

4.3 MomDeaccess(AQid); Message Sending - MomSend()

The MomSend() verb is used for sending a message. As demonstrated in the simple application presented in Chapter 3, the basic utilization of MomSend() is straight-forward. We will now review the arguments to MomSend() by way of a second example.

Consider the following MomSend() call:

```
/*
 * Send a message to app-queue "abc".
 */

XINT        TargetAQid;
MOM_MSGID   RetMsgId;
CHAR        *MsgText = "hello world";
XINT        MsgLen = 12;

TargetAQid = MomAccess("@abc");

MomSend(
        TargetAQid,             /* Handle of target AQid */
        MsgText,                /* Message being sent */
        MsgLen,                 /* Size (in bytes) of message */
        MOM_PRIORITY_NORMAL,    /* Priority of sent message */
        MOM_TRACK_DELIVERED,    /* Track message till it is received */
        MOM_REPLY_NONE,         /* No reply expected */
        &RetMsgId,              /* Message-id assigned to sent message */
        MOM_WAIT                /* Block if system is busy */
      );
```

The MomSend() verb takes eight basic arguments and one optional argument. We will now review the basic arguments in the context of the above example. A subsequent example, describing client/server communication, will demonstrate the optional argument to MomSend().

- **`TargetAQid`** defines the AQid of the app-queue being targeted. Typically (as in the above case), the value for **`TargetAQid`** is acquired via a prior call to MomAccess().
- **`MsgText`** is a pointer to the message buffer being sent. Messages sent by MomSend() can be in any form – text, structures, images, etc.– and are not interpreted by MomSend().
- **`MsgLen`** is the length of data bytes to be sent by the MomSend() call.
- **`MOM_PRIORITY_NORMAL`** directs *X•IPC* to send the message with a "normal" priority. There are a wide range of possible priority values that may be specified. In addition to normal priority, two of the other more common values are: `MOM_PRIORITY_HIGH` and `MOM_PRIORITY_LOW`. (Refer to Appendix B, Message Priority Specification, for additional details on priority specification.)
- **`MOM_TRACK_DELIVERED`** directs *X•IPC* to track the message being sent until it is dequeued from the target AQid. Other tracking level values may be specified. (Refer to Appendix A, Message Status and Tracking Levels, for additional details on message tracking levels.)
- **`MOM_REPLY_NONE`** alerts *X•IPC* to the fact that no return message is expected in response to this message being sent. We will see later, in the discussion of client/server inquiry-response communication, that this MomSend() argument may be used to correlate inquiry and response messages.
- **`&RetMsgId`** is a pointer to a variable of type `MOM_MSGID`. It is returned populated with message-id data about the message sent. We will see that a message-id is an important tool for message tracking and inquiry-response correlation. Alternatively, `NULL` could have been specified.
- **`MOM_WAIT`** is one of the six *X•IPC* blocking options. It directs *X•IPC* to block, if necessary, when submitting the message into the MomSys subsystem. This will occur if the subsystem is momentarily congested at the time of the call.

4.3.1 OPTIONAL ARGUMENTS TO MOMSEND()

The Reference Manual pages for MomSend() lists a number of optional arguments that may be specified as part of a call to MomSend(). These optional arguments are useful for accomplishing various objectives when sending a message. Note that these optional arguments are inserted in the argument list prior to the call's blocking option. See the `MOM_EXPIRE()` example below.

4.3.1.1 The MOM_REPLYTO() Option

One of the more important optional arguments to MomSend() is the `MOM_REPLYTO(`*`MsgId`*`)` argument. We will see later, in the discussion of client/server inquiry-response communication, that this argument may be used for correlating a response message back to its particular inquiry messages.

4.3.1.2 The MOM_EXPIRE() Option

Another optional argument is the `MOM_EXPIRE(`*`TimeLimit`*`)` argument. Using this argument, a programmer can establish an expiration time for the particular message being sent. The *`TimeLimit`* value is treated as an integer number of seconds. This value overrides other instance-defined expiration time-limits.

```
/*
 * Send a message to app-queue "abc" having an expiration time-limit
 * of one hour (3600 seconds).
 */

XINT        TargetAQid;
MOM_MSGID   RetMsgId;
CHAR        *MsgText = "hello world";
XINT        MsgLen = 12;

TargetAQid = MomAccess("@abc");

MomSend(
        TargetAQid,             /* Handle of target AQid */
        MsgText,                /* Message being sent */
        MsgLen,                 /* Size (in bytes) of message */
        MOM_PRIORITY_NORMAL,    /* Priority of sent message */
        MOM_TRACK_DELIVERED,    /* Track message till it is delivered */
        MOM_REPLY_NONE,         /* No reply expected */
        &RetMsgId,              /* Message-id assigned to sent message */
        MOM_EXPIRE(3600),       /* Msg expires in one hour (3600 secs) */
        MOM_WAIT                /* Block if system is busy */
      );
```

Messages that do not reach their tracking-level state within their expiration time-limit are automatically expired. Depending on instance configuration parameters, such messages are then either journaled or deleted from *X•IPC* without a trace. Refer to Appendix A, Message Status and Tracking Levels, for a more detailed discussion of message tracking.

4.3.2 BLOCKING OPTIONS

The last argument in the MomSend() call is the function's "blocking option" argument. *X•IPC* blocking options define what an API function does *when it can't immediately complete at the time of the call*. *X•IPC* provides a range of six possible blocking options for all verbs that have the potential to block. (For a complete discussion of this topic refer to the section on *X•IPC* Blocking Options in the *X•IPC* User Guide.)

It is important to understand what the phrase "can't immediately complete at the time of the call" means in the context of the MomSend() function call. This requires a brief description of how messages in general move in the MomSys subsystem of *X•IPC* .

A call to MomSend() dispatches a message into the caller's local instance. The message is then moved forward towards its target app-queue as fast as possible. Of course, if the network is down or the remote node is down, or the targeted app-queue is not around, then X•IPC is responsible to move the message forward as these impediments become corrected.

The blocking option argument to MomSend() deals with the *first* part of the process: submitting the message into the local instance environment. If the local instance is capable of accepting the message immediately, then MomSend() is said to have been able to "complete immediately" at the time of its call. Hence, in such cases the blocking option is not employed. This should be the typical behavior in a properly configured local instance. In such settings, calls to MomSend() should be forced to block only occasionally, if at all.

When the local instance is congested to the extent that its internal resources do not permit the local instance to accept the message being submitted by MomSend(), then the MomSend() call will "block" as designated by the specified blocking option until the local instance can accept the message.

The above example is coded with the `MOM_WAIT` blocking option, indicating that the caller wishes the MomSend() verb to block if necessary before submitting the message into the local instance and returning control to the user.

4.3.3 OPTIONAL FLAGS TO MOMSEND()

The Reference Manual definition of MomSend() describes the optional flags that may be specified as part of a call to MomSend(). (The flags should be ORed to the *left* of the blocking option argument.)

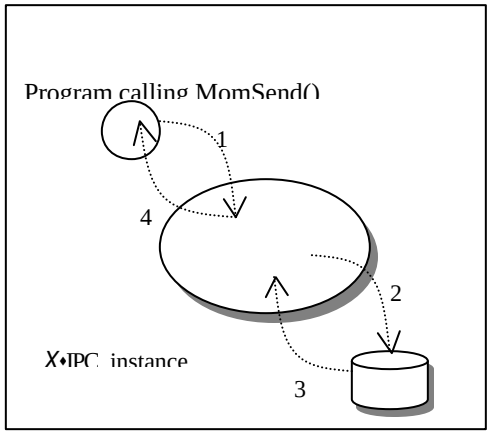
4.3.3.1 The MOM_FASTPATH Option

The `MOM_FASTPATH` optional flag allows the user to specify to X•IPC that the current MomSend() operation should be completed (i.e., sending the specified message into X•IPC and returning control to the user) *without* synchronizing the message's data to disk. This has the advantage of increasing the performance of such MomSend() operations, but has the disadvantage that messages sent with such a flag are not recoverable following a system failure.

The following diagrams illustrate the difference between using `MOM_FASTPATH` and not.

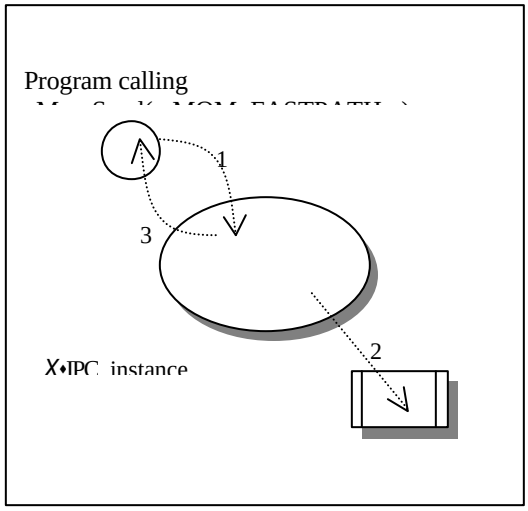
Steps of execution when `MomSend()` is called without `MOM_FASTPATH`:

1. User's message is submitted to the X•IPC instance.
2. Instance initiates the writing of message data to the disk.
3. Instance is notified that message is safely on the disk.
4. User's `MomSend()` call completes execution.



Steps of execution when `MomSend()` is called with `MOM_FASTPATH`:

1. User's message is submitted to the *X•IPC* instance.
2. Instance writes message data to fast, but volatile, memory.
3. User's `MomSend()` call completes execution.



4.3.3.1.1 Performance Considerations

`MOM_FASTPATH` can be specified on a *per-message* basis. This allows the application developer to decide, at run-time, which messages require synchronous disk updating and which do not. Synchronization of an instance's Message Repository to disk occurs at the instance level. As long as MomSend() calls specify `MOM_FASTPATH`, message data is stored in fast but volatile memory, either in RAM or on disk, depending on the operating system's own needs. Issuing calls to MomSend() without `MOM_FASTPATH`, by contrast, forces the operating system to perform a disk flush of message data as part of each MomSend() call.

Mixing such calls within a single instance will produce mixed results with regard to instance performance. The greater the relative number of MomSend() operations specifying `MOM_FASTPATH`, the better the overall instance performance. The inverse is true as well.

4.3.3.2 The MOM_RETURN Option

The `MOM_RETURN` option, which is only valid when accompanying one of the three asynchronous blocking options, directs *X•IPC* to complete the operation synchronously if there is no need to "block" (e.g., the desired message is on the app-queue) and to "go asynchronous" only if the operation cannot be completed immediately. (Refer to the section on *X•IPC* Blocking Options in the *X•IPC* User Guide for a detailed discussion of this option.)

4.4 Message Receiving - MomReceive()

The MomReceive() verb is used for receiving a message from an app-queue. As demonstrated in the earlier simple application, the basic utilization of MomReceive() is straight-forward. We now review the arguments to MomReceive() by way of a second example.

Consider the following call:

```
/*
 * Receive message from app-queue "abc".
 * /

XINT        SourceAQid, MsgLen;
XINT        InBufLen = 16;
CHAR        InBuf[16];
MOM_MSGID   RetMsgId;
MOMINFOMSG  RetInfoMsg;

SourceAQid = MomCreate("abc", MOM_APPQUEUE_DISK_REGISTER);
```

```
MsgLen = MomReceive(
            SourceAQid,             /* Handle of source AQid */
            InBuf,                  /* Buffer for receiving message */
            InBufLen,               /* Size (in bytes) of receive buffer */
            MOM_MESSAGE_FIRST,      /* Request first (front) message on app-queue */
            &RetReplyAQid,          /* AQid of app-queue to send response msg */
            &RetMsgId,              /* Message-id of received message */
            &RetInfoMsg,            /* Detailed info on received message */
            MOM_WAIT                /* Block if system is busy */
      );
```

MomReceive(), when successful, returns the length (in bytes) of the message that is received. MomReceive() always accepts eight arguments. In the context of the above example, they are:

SourceAQid defines the AQid of the app-queue being received from. This app-queue *must* be within the caller's local instance. Typically (as in the above case), the value for **SourceAQid** is acquired via a prior call to MomCreate().

InBuf is a pointer to the receiving message buffer.

InBufLen is the length (in bytes) of the receiving buffer (i.e., **InBuf** in the above example)

MOM_MESSAGE_FIRST directs X•IPC to retrieve the "first" message from app-queue "abc". When created, app-queue "abc" employs a default *natural* sequencing of messages that is based on message arrival time. Thus, the specification of **MOM_MESSAGE_FIRST** returns the first (i.e., oldest) message in that sequence. **MOM_MESSAGE_FIRST** is one of the message-specifiers (known as *MsgSpecifier*) that are predefined by X•IPC for supporting different message-selection scenarios that can arise. These are listed below. Moreover, X•IPC allows a user to define his own customized message specifiers (Refer to Appendix C for more detailed information on Advanced Message Selection.)

&RetReplyAQid is a pointer to an integer variable. It is returned populated with an AQid where a response message should be sent. Alternatively, it may be returned populated with the value MOM_REPLY_NONE, indicating that no reply-AQid was stipulated by the sender of the message. The NULL pointer may be passed if no reply AQid is desired.

&RetMsgId is a pointer to a variable of type MOM_MSGID. It is returned populated with message-id data about the received message. The NULL pointer may be passed if no message-ID is desired.

&RetInfoMsg is a pointer to a variable of type MOMINFOMSG. This structure is returned populated with extended data about the returned message (*who sent it, when it was sent, where it was sent from, etc.*). The NULL pointer may be passed if no extended message data is desired.

MOM_WAIT is one of X•IPC 's six blocking options. It directs X•IPC to block, if necessary, when retrieving the message from the MomSys subsystem. This will occur if the app-queue is empty or if the specified message is not on the app-queue at the time of the call.

4.4.1 MESSAGE SELECTION

Message specification is accomplished via the *MsgSpecifier* argument to MomReceive(). *MsgSpecifier* identifies which message is to be retrieved from *SourceAQid*. Proper utilization of this argument requires a basic understanding of how messages reside on an app-queue.

When an app-queue is created one of its defining attribute specifies the *natural* sequencing of messages on that app-queue. (Refer to the MomAttrSet() function call for details on app-queue attribute specification.) An app-queue has one of the following attributes: natural sequencing by *Time* or natural sequencing by *Priority*.

Message specification semantics can differ depending on the natural sequencing of messages on an app-queue. For example, specifying the "first" message on an app-queue means the "oldest" message if the natural sequencing is by *Time;* it means the "highest" priority message if the natural sequencing is by *Priority*.

The following are the *MsgSpecifier* values that are provided:

MOM_MESSAGE_FIRST	Retrieve the first message from natural sequence. If *Time,* the oldest message is returned. If *Priority,* the highest priority message is returned.
MOM_MESSAGE_LAST	Retrieve the last message from natural sequence. If *Time,* the newest message is returned. If *Priority,* the lowest priority message is returned.
MOM_MESSAGE_NEXT(*MsgId*)	Retrieve the next message from within natural sequence following the message identified by *MsgId.**. If *Time*, the next oldest message is returned. If *Priority*, the next highest priority message is returned.
MOM_MESSAGE_PREV(*MsgId*)	Retrieve the previous message from within natural sequence following the message identified by *MsgId.** If *Time*, the previous oldest message is returned. If *Priority*, the previous highest priority message is returned.
MOM_MESSAGE_DIRECT(*MsgId*)	Retrieve the message identified by *MsgId*. *
MOM_MESSAGE_DIRECT_RMT (RmtNode, RmtInstance, RmtMsgId)	Retrieve a message based on its Remote identification: RmtNode is name of sender node RmtInstance is name of sender instance RmtMsgId is the MsgId that was assigned to the message when it was sent via the sender instance
MOM_MESSAGE_REPLYTO(*MsgId*)	Retrieve the response message to the request message that was previously sent by MomSend() and identified as *MsgId*.

(* Note: The message represented by MsgId, where indicated with an asterisk, must still be on the app-queue at the time of the MomReceive() call. This is typically accomplished by having performed an earlier call to MomReceive() in which the MOM_NOREMOVE flag was set. The MsgId returned from that call can serve as the "cursor" for subsequent MomReceive() calls.)

The above listed values for *MsgSpecifier* are actually macros that are based on a more general syntax of message specification. Refer to Appendix C, Message Specification in MomReceive(), for details of this syntax.

4.4.2 OPTIONAL FLAGS TO MOMRECEIVE()

The Reference Manual definition of MomReceive() lists a number of optional flags that may be specified as part of a call to MomReceive(). MomReceive() takes the same blocking options as MomSend() (See section 4.3.3); the optional flags should be ORed to the left of the blocking option. These flags are:

- `MOM_NOREMOVE`
- `MOM_RETURN`

The following sections briefly describe these flags. Refer to the MomSys Reference Manual definition for further details.

4.4.2.1 The MOM_NOREMOVE Option

The `MOM_NOREMOVE` flag allows an application program to receive a "copy" of a message from an app-queue, but leaves the message remaining on the app-queue. This form of message previewing, when combined with the "navigational" message-specifiers (e.g., next, previous, etc.) facilitates development of applications that can browse and examine sequences of app-queue messages.

Example:

```
/*
 * Receive message copy but leave actual message still on app-queue.
 * /

MsgLen = MomReceive(
            SourceAQid,                 /* Handle of source AQid */
            InBuf,                      /* Buffer for receiving message */
            InBufLen,                   /* Size (in bytes) of receive buffer */
            MOM_MESSAGE_FIRST,          /* Request first message on app-queue */
            &RetReplyAQid,              /* AQid of app-queue to send response msg */
            &RetMsgId,                  /* Message-id of received message */
            &RetInfoMsg,                /* Detailed info on received message */
            MOM_NOREMOVE | MOM_WAIT     /* Don't remove msg. Block if necessary */
        );
```

Refer to the MomSys Reference Manual definition for details on employing the MOM_NOREMOVE flag.

4.4.2.2 The MOM_RETURN Option

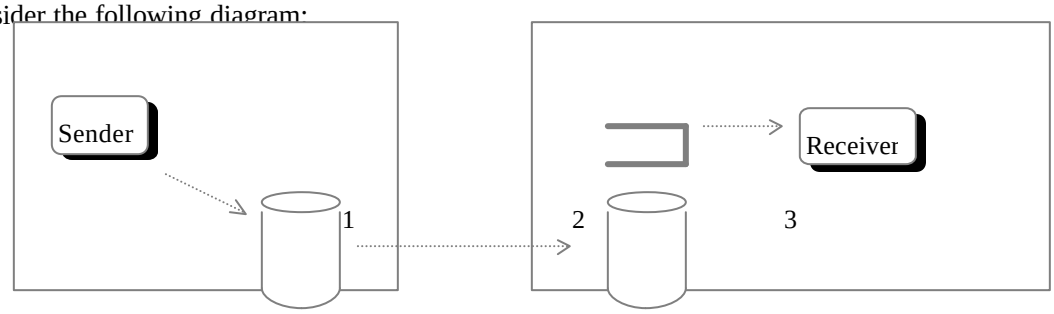
The MOM_RETURN option, which is only valid when accompanying one of the three asynchronous blocking options, directs X•IPC to complete the operation synchronously if there is no need to "block" (e.g., the desired message is on the app-queue) and to "go asynchronous" only if the operation cannot be completed immediately. (Refer to the section on X•IPC Blocking Options in the X•IPC User Guide for a detailed discussion of this option.)

4.5 Message Tracking

As was shown in the above coding examples, the MomSend() function requires that a tracking-level argument be defined for each message that is sent. This argument to MomSend() has a great influence on how far the message is tracked by X•IPC in the course of its trip.

A general understanding of message movement within the X•IPC MomSys programming model is a prerequisite to proper utilization of the subsystem.

Consider the following diagram:



4.5.1 MESSAGE STATUS VALUES

An *X•IPC* MomSys message goes through three well-defined, trackable stages as it moves from sender to receiver program. These stages are identified numerically in the above diagram. The message status values that correspond to these stages are:

MOM_STATUS_HELD	Message is currently in the sender's message repository, but has not yet been shipped to the receiver node.
MOM_STATUS_SHIPPED	Message has shipped to the receiver's message repository and has been logically inserted within the targeted app-queue, but it has not been received and removed by a receiving program.
MOM_STATUS_DELIVERED	Message has been received and removed from the app-queue by a receiving program.

Two additional pseudo-status values that are occasionally employed within MomSys are:

MOM_STATUS_COMPLETE	Message status has achieved the tracking-level that was specified for it when the message was sent via MomSend().
MOM_STATUS_INCOMPLETE	Message status has not yet achieved the tracking-level that was specified for it when the message was sent via MomSend().

MomEvent() is an example of a function that employs the `MOM_STATUS_COMPLETE` for creating an event that occurs when a given message reaches the tracking-level that it was sent with. Refer to the description of MomEvent()for details.

4.5.2 MESSAGE TRACKING LEVELS

Just how far a message is actually tracked by *X•IPC* is a function of the tracking-level that is specified within the MomSend() verb when the message is sent. The two message tracking levels that may be specified are:

MOM_TRACK_SHIPPED	Track the message being sent until it has attained the status of `MOM_STATUS_SHIPPED`.
MOM_TRACK_DELIVERED	Track the message being sent until it has attained the status of `MOM_STATUS_DELIVERED`.

Note that a message status is updated in the sender's message repository up to the level requested by the tracking level argument of the MomSend() function - *but no further*.

Thus, a message sent with a tracking-level of `MOM_TRACK_SHIPPED` is tracked up to the point that the message attains a status of `MOM_STATUS_SHIPPED`, after which point no further tracking is performed. A more complete description of message tracking is presented in Appendix A, Message Status and Tracking Levels.

4.6 Client/Server Interaction

Critical to utilizing a message-oriented middleware technology for developing large scalable client/server applications is the ability to support massive client population deployments whose composition is in a continuous state of flux, and to do so without requiring any server-side involvement (e.g., reconfiguration, table definitions, etc.). This is readily achievable using MomSys.

MomSys allows a server application to receive inquiry messages from a large population of client programs and to send response messages to each and every respective client *without* the need to know who, or where, the inquiring clients are.

4.6.1 REQUEST-RESPONSE PROGRAMMING STEPS

The following steps summarize what occurs during a typical client/server request-response exchange of messages. A coding example is presented in the next section.

A client creates a private app-queue (i.e., created with name `MOM_PRIVATE`) within its local instance for receiving response messages. There is no need for each client to uniquely name its response application queue, nor is there a need to register the app-queue in the catalog.

The client sends an inquiry message to a server via a call to MomSend() in which it specifies the AQid of its private app-queue as the reply-AQid where it expects to receive a response message.

The server receives the inquiry message via a call to MomReceive() and with it is given the message-ID of the received inquiry message, as well as the reply-AQid of the response app-queue (i.e., the AQid of the client's private app-queue).

The server processes the message and sends a response message to the client via a call to MomSend() by specifying the client's private app-queue AQid as the target, and by specifying the `MOM_REPLYTO(`*`MsgId`*`)` option, where *`MsgID`* identifies the received inquiry message. This option causes the response message being sent to correlate with the client's original inquiry message.

The client issues a MomReceive() call on its private app-queue to receive the response message to its inquiry message. The MomReceive() call specifies the `MOM_MESSAGE_REPLYTO(`*`MsgId`*`)` message-specifier, where *`MsgID`* identifies the client's originally sent message, so that it receives the response message sent by the server.

It is important to note that, in this manner, a client may issue multiple inquiry messages to multiple unrelated servers where all response messages are directed to arrive on the client's single private app-queue. Using the `MOM_MESSAGE_REPLYTO()` MomReceive() message-specifier, the client can selectively retrieve the responses to its outstanding inquiries in the order that it wants them. (This will be elaborated on later in section 4.6.4 on Inquiry-Response Correlation.)

Furthermore, client-sent messages that arrive at a server may be responded to by the server without any awareness of the location or identification of the originating client.

4.6.2 CLIENT-SIDE PROGRAMMING EXAMPLE

```
/*
 * Client program "client". Note, error checking is omitted
 * to enhance program readability.
 */

#include "xipc.h"

VOID
main( argc, argv )
```

```
XINT argc;
CHAR **argv;

{
    XINT ServerQ_AQid, ReplyQ_AQid;
    MOM_MSGID RequestMsgId;
    CHAR InBuffer[64];
    XINT InBufferLen = 64;

    /*
     * Log in to client's local instance I1,.. then ...
     * Create client's private app-queue, .... then ...
     * Access handle to the server app-queue "ServerQ"
     */

    XipcLogin("@I1", "client");
    ReplyQ_AQid  = MomCreate(MOM_PRIVATE, MOM_APPQUEUE_DISK);
    ServerQ_AQid = MomAccess("@ServerQ");

    /*
     * Send request message to ServerQ. Note that the returned message-id
     * value (returned within RequestMsgId) will be used in the next
     * step for requesting a response to original request.
     */

    MomSend(
            ServerQ_AQid,           /* AQid of app-queue "ServerQ" */
            "hello world",          /* Message being sent */
            12L,                    /* Size (in bytes) of message */
            MOM_PRIORITY_NORMAL,    /* Priority of sent message */
            MOM_TRACK_DELIVERED,    /* Track msg until received by server */
            ReplyQ_AQid,            /* AQid of client's reply app-queue */
            &RequestMsgid,          /* Message-id of inquiry msg being sent */
            MOM_WAIT                /* Block if system is busy */
          );

    /*
     * Receive response message.
     */

    MomReceive(
            ReplyQ_AQid,                /* AQid of client's private app-queue */
            InBuffer,                   /* Buffer to accept reply message */
            InBufferLen,                /* Size (in bytes) if InBuffer */
            MOM_MESSAGE_REPLYTO(RequestMsgId),   /* Select to receive reply msg */
                                                 /* to sent request msg */
            NULL,                       /* (we don't expect a ReplyAQid) */
            NULL,                       /* (we don't need MsgId of reply) */
            NULL,                       /* (we don't need detailed Msg Info ) */
            MOM_WAIT                    /* Block until reply message arrives */
           );

    printf("got reply message: %s\n", InBuffer);
    XipcLogout();
}
```

4.6.3 SERVER-SIDE PROGRAMMING EXAMPLE

```
/*
 * Server program “server”. Note, error checking is omitted
 * to enhance program readability.
 */

#include “xipc.h”

VOID
main( argc, argv )
XINT argc;
CHAR **argv;
{
    XINT ServerQ_AQid, ReplyQ_AQid;
    MOM_MSGID RequestMsgId;
    CHAR Buffer[64], *p;
    XINT BufferLen = 64;

    /*
     * Log in to server’s local instance I2, then ...
     * Create server app-queue “ServerQ”. Note that the
     * MOM_APPQUEUE_DISK_REGISTER argument to MomCreate()
     * causes the created app-queue to be registered automatically.
     */

    XipcLogin(“@I2”, “server”);
    ServerQ_AQid = MomCreate(“ServerQ”, MOM_APPQUEUE_DISK_REGISTER);

    /*
     * Receive request message from client.
     */

    MomReceive(
            ServerQ_AQid,               /* Recv msg from ServerQ */
            Buffer,                     /* Buffer to accept request message */
            BufferLen,                  /* Size (in bytes) of buffer */
            MOM_MESSAGE_FIRST,          /* Select to receive first msg on app-queue */
            &ReplyQ_AQid,               /* Set with AQid to send response to */
            &RequestMsgId,              /* Set with MsgId of request message */
            NULL,                       /* (we don’t need detailed Msg Info ) */
            MOM_WAIT                    /* Block until reply message arrives */
           );

    /*
     * Process the request message. Our “fancy” server takes a null-terminated string
     * sent by the client, changes all characters to upper case and returns the
     * modified string to the client.
     */

    p = Buffer;
    while (*p++)
        toupper(*p);


    /*
     * Send response message to originating client. Note that the request
     * msg’s message-id value (returned within RequestMsgId) is used
     * for correlating response to original client requester.
     */
```

```
    MomSend(
            ReplyQ_AQid,            /* Handle of client's private app-queue */
            Buffer,                 /* Message being sent */
            strlen(Buffer)+1,       /* Size (in bytes) of response message */
            MOM_PRIORITY_NORMAL,    /* Priority of sent message */
            MOM_TRACK_SHIPPED,      /* Track msg till shipped & stored in rmt node */
            MOM_REPLY_NONE,         /* AQid of client's reply app-queue */
            NULL,                   /* (we don't need MsgId of reply) */
            MOM_REPLYTO(RequestMsgid),   /* Direct XIPC to correlate this message */
                                         /* as a response to the original request msg.*/
            MOM_WAIT                     /* Block if system is busy */
          );

    XipcLogout();
}
```

4.6.4 REQUEST-RESPONSE CORRELATION

The issue of inquiry-response message correlation becomes important in situations where a client wishes to send multiple inquiries "in flight" simultaneously to multiple servers, and to subsequently receive the response messages in a *specific* sequence. Because the client does not know the order in which response messages will arrive on its private response queue, it cannot simply ask for the "next" message. In such a situation, the message specifier `MOM_MESSAGE_REPLYTO(`***`RequestMsgId`***`)` is critical in selecting the specific response that is desired.

Thus, if the client performed three MomSend() calls that sent three request messages to three servers, it would save the three message-ids returned from the MomSend() calls and subsequently specify them as part of the `MOM_MESSAGE_REPLYTO()` message-specifier in the MomReceive() calls that received the responses.

Consider the following client code segment:

```
main(...)
...
{
    /*
     * Issue three requests to three servers.
     * Returned message-ids are saved.
     */

    MomSend (Server1_AQid, ..., &RetMsgId1, ...);
    MomSend (Server2_AQid, ..., &RetMsgId2, ...);
    MomSend (Server3_AQid, ..., &RetMsgId3, ...);



    /*
     * Receive the three response messages in the
     * reverse order from which the requests were sent.
     * Returned message-ids are used for this purpose.
     */

    MomReceive ( ReplyAQid, ..., MOM_MESSAGE_REPLYTO(RetMsgId3), ...);
    MomReceive ( ReplyAQid, ..., MOM_MESSAGE_REPLYTO(RetMsgId2), ...);
    MomReceive ( ReplyAQid, ..., MOM_MESSAGE_REPLYTO(RetMsgId1), ...);
}
```

Its important to note that there is no special server-side logic needed to support this form of client-side activity so long as the servers are coded to handle inquiry and response messages in the general manner described in the prior examples.

5. BASIC MOMSYS CONFIGURATION AND ADMINISTRATION

5.1 The *X•IPC* Platform Environment

Recall from the *X•IPC* User Guide that any platform supporting *X•IPC* activity must first initialize the "*X•IPC* Platform Environment" on that platform. This topic is described in detail in the *X•IPC* User Guide.

Accordingly, application programs that employ the *X•IPC* MomSys subsystem (or any part of *X•IPC* for that matter) *cannot* be run until the *X•IPC* platform environment has been started on that platform. As described in the *X•IPC* User Guide, the utility command for starting an *X•IPC* platform environment is `xipcinit`; and for terminating the environment, `xipcterm`. The `xipcinit` command reads its configuration parameters from the *X•IPC* Platform Environment File (i.e., `xipc.env`). This file directs `xipcinit` as to what form (i.e., capacity) the platform environment will take; as well as what *X•IPC* namespaces are to be supported.

One of the primary functions of the `xipcinit` command in starting up a platform's *X•IPC* environment is to start up the platform's *X•IPC* namespace catalog server. The next few sections examine the topic of catalog server configuration. Refer to the *X•IPC* User Guide for a more general discussion of the *X•IPC* Environment Platform and the means for invoking `xipcinit` and `xipcterm`.

Before actually listing and defining the `xipc.env` parameters that are related to the catalog server, it is useful to step back and review the general topic of *X•IPC* namespaces from the catalog server perspective. Along the way we will also see how *X•IPC* instances affiliate themselves with an existing namespace.

(The following discussion references certain configuration parameters that are required for catalog and instance configuration. These examples are presented for the purpose of describing configuration concepts. Tables containing the complete list of parameters and their possible and default values are presented towards the end of the section.)

5.2 Establishing a Namespace

We have thus far described *X•IPC* namespaces as being "*somehow*" managed by *X•IPC* catalog server programs, but we have not gone into detail as to how catalog servers are configured to perform this function.

Within a typical *X•IPC* environment a *single* catalog server program is present on *each* network node. The catalog server programs active on a network perform a range of *X•IPC* namespace related work, including:

Support the abstraction of *X•IPC* namespace location transparency

Support actual namespace data (usually on a small subset of the network's nodes)

Perform dynamic namespace discovery functions

Support catalog data redundancy for handling catalog fail-over and recoverability

5.2.1 NAMESPACE CONFIGURATION

The primary function of the *X•IPC* catalog server programs is to support the abstraction of *X•IPC* namespaces "*spread over the network*". Based on this abstraction, programs are able to access app-queues network-wide, without concern for their location.

In fact, beneath this abstraction, actual data for each existing *X•IPC* namespace must physically reside on some set of network nodes. These nodes are referred to as the "anchor nodes" for the namespace.

5.2.1.1 Namespace Definition

A namespace is established by defining the set of nodes upon which the namespace's data will physically reside, i.e. its anchor nodes. This set is specified within the `xipc.env` files of the anchor nodes, as well as other nodes that are to have access to the namespace, via the `NAMESPACE` statement, as follows:

Syntax:

```
[CATALOG.protocol]
NAMESPACE   namespace-name:node-list
```

where:

- `[CATALOG.protocol]` is the section header for catalog parameters specific to a particular network protocol. TCP/IP is currently the only supported protocol.
- *`namespace-name`* identifies the specific namespace being defined.
- *`node-list`* identifies the network nodes that will serve as anchor nodes for the namespace.

Example:

```
[CATALOG.TCPIP]
NAMESPACE    xyz:Server1,Server2          # Defines namespace "xyz".
                                          # "xyz" will be anchored
                                          # in replicated form on nodes
                                          # "Server1" and "Server2"
```

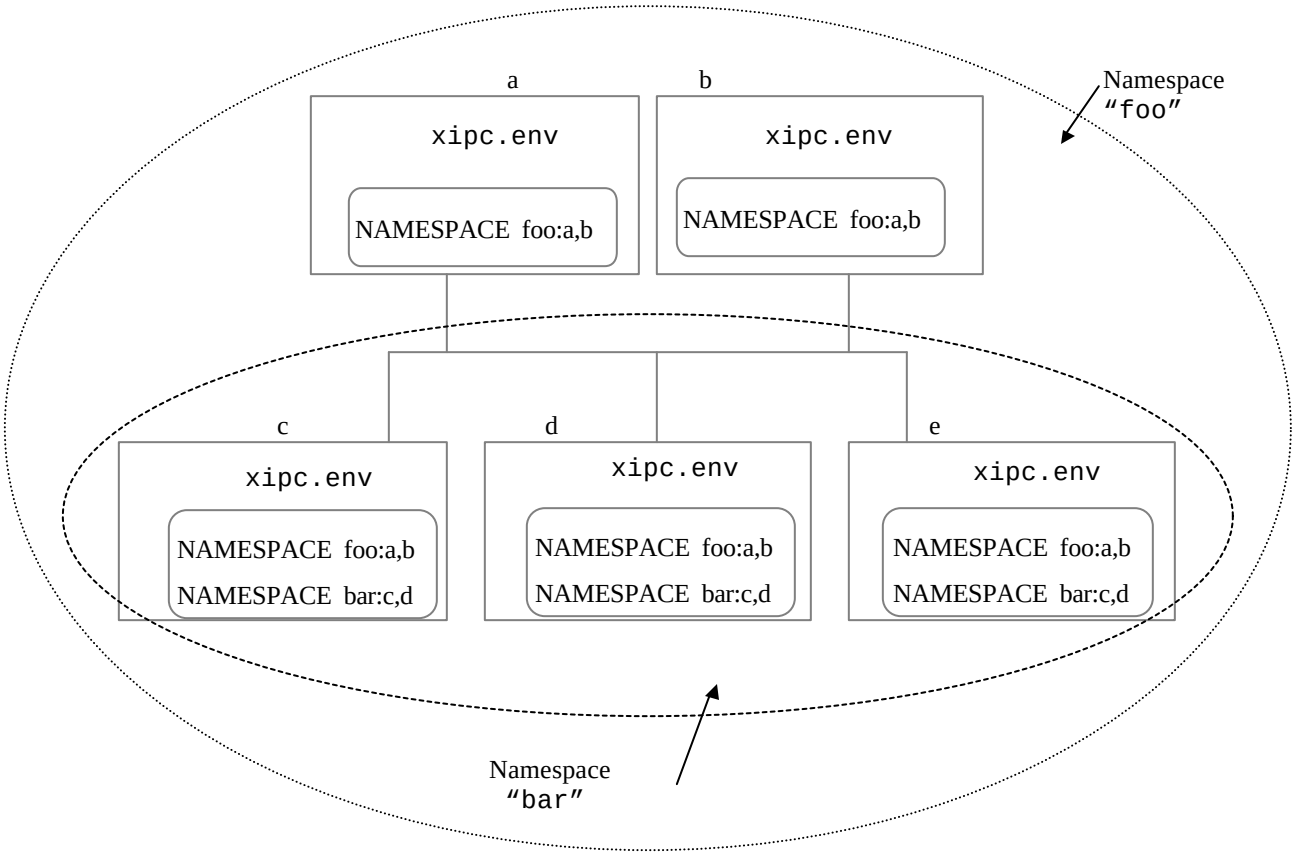
The above statements define namespace "`xyz`" as being anchored, in replicated form, on platforms `Server1` and `Server2`. Other network nodes planning to reference the namespace must specify that intent by including the namespace statement within their `xipc.env` files, as well:

5.2.1.2 Namespace Configuration Example

Consider the following five-node network (nodes are named: `a`, `b`, `c`, `d`, `e`), on which we would like to define two *X♦IPC* namespaces "`foo`" and "`bar`", where:

- Namespace `foo` is to be anchored on nodes `a` and `b`, and is to be accessible from any of the nodes on the network, and
- Namespace `bar` is to be anchored on nodes `c` and `d`, but is to be only accessible from nodes `c`, `d` and `e`.

The following diagram depicts our network, as well the NAMESPACE statements to be inserted within the respective xipc.env platform configuration files for creating the desired namespace environments.



Namespace foo is anchored on nodes a and b, and is generally accessible from the entire network, This is accomplished by inserting in each platform's xipc.env file the following NAMESPACE statement:

```
NAMESPACE    foo:a,b
```

Namespace bar is anchored on nodes c and d, and is only accessible from nodes c, d and e. This is accomplished by inserting within the platform's xipc.env files on nodes c, d and e the following NAMESPACE statement:

```
NAMESPACE    bar:c,d
```

By *not* including this statement in the xipc.env files of nodes a and b we have made namespace bar inaccessible from instances on those nodes; that is, they cannot affiliate with namespace bar. Instance affiliation to namespaces is described in the next section.

5.3 X♦IPC Instance Namespace Affiliation

Once an X♦IPC namespace has been defined over a network, it becomes possible for X♦IPC instances to affiliate themselves with the namespace. As described earlier, the access point for a program to reach an X♦IPC namespace is via an X♦IPC instance. A program *must* first log into an X♦IPC instance (via the XipcLogin() function call) in order to access an X♦IPC namespace. The namespace that it accesses is the namespace that the instance is "affiliated" with.

An X♦IPC instance can affiliate itself with at most *one* X♦IPC namespace. An instance establishes its affiliation with a namespace by declaring so within its instance configuration (.cfg) file, by including the following `NAMESPACE` parameter statement within the `[XIPC]` section of that file, as follows:

Syntax:

```
[XIPC]
NAMESPACE    namespace-name
```

where, *namespace-name* identifies the specific X♦IPC namespace with which the instance will be affiliated.

Example:

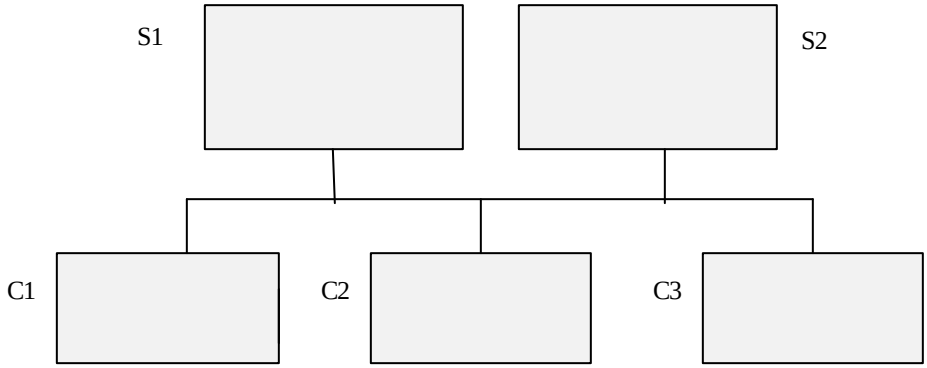
```
[XIPC]
NAMESPACE    foo          # Affiliate instance with namespace "foo"
```

An instance configuration file that does *not* have a `NAMESPACE` statement will, when started by `xipcstart`, create an instance that is not affiliated with any X♦IPC namespace.

5.4 X♦IPC Configuration: A Client/Server Example

We will now apply these namespace configuration concepts to a typical client/server example; one in which scalability and simplicity of configuration are critical. Consider an application, having the following requirements:

- A TCP/IP network is to be set up as a client/server environment.
- Two nodes will support server programs. We will call these nodes S1 and S2. Server programs may execute on either S1 or S2. Which server programs are running on which server platforms may change from day to day, and should therefore be dynamically configured at run-time, without client awareness.
- There will initially be three client nodes – C1, C2 and C3 – but the population of client nodes will vary widely, with new clients being added and old ones deleted on a continuous basis. The environment must be able to support these changes dynamically.

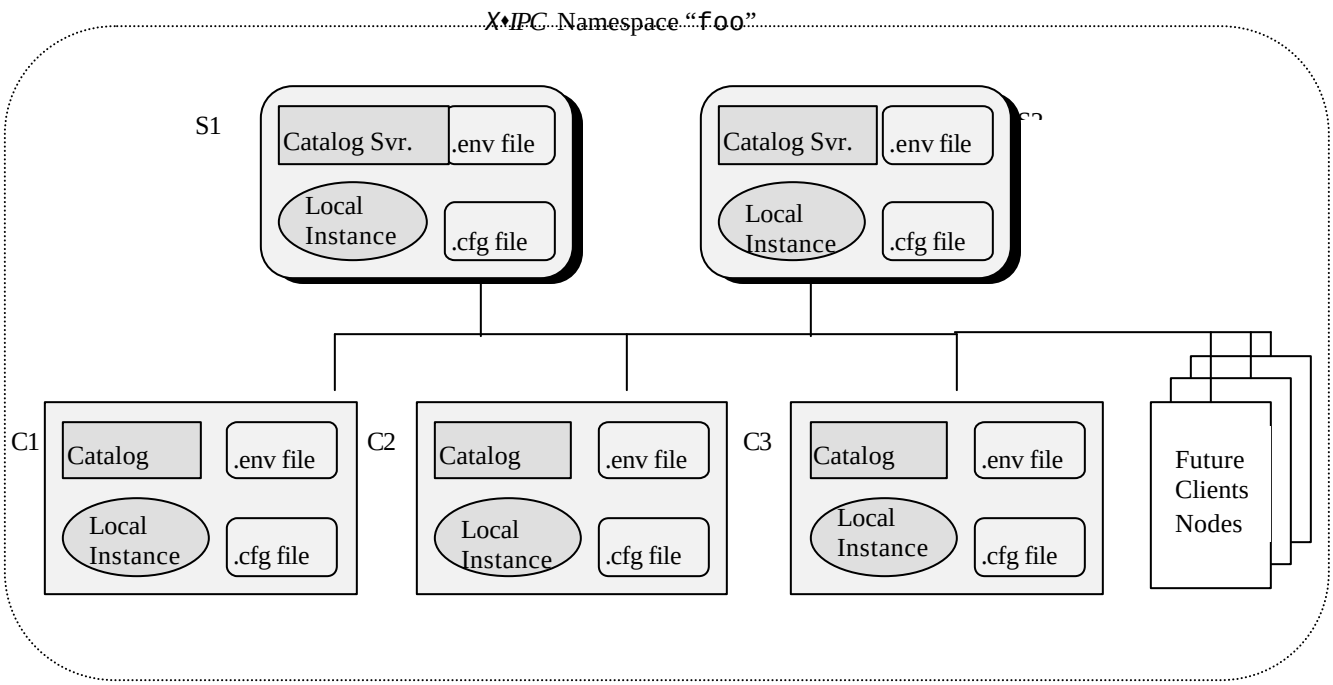


- Programs running – clients and servers – must be able to exchange asynchronous, guaranteed-delivery messages in a network-transparent and scalable manner.

5.4.1 AN X♦IPC SOLUTION

We will now develop a high-level solution to the above set of requirements where *X♦IPC* MomSys is employed as the messaging infrastructure. The main focus of this exercise will be to study how platform and instance namespace configuration is employed to address the application's scalability needs.

Applying *X♦IPC* MomSys to the above requirements yields the following configuration components:



The following points describe the above set of platform catalog and instance configurations:

An *X♦IPC* namespace "`foo`" will be established on the network. Namespace "`foo`" will be accessible from *all* points on the network.

Namespace "`foo`" data will physically be maintained - anchored - on servers S1 and S2 in a replicated manner. This will provide a high-degree of availability of the namespace so that in the event that either server fails, the namespace "`foo`" will survive. Namespace redundancy and fail-over activity will be transparent to all clients (and servers).

Server programs will create app-queues that have well-known names and that will be registered in namespace "`foo`". Clients will send messages to these app-queues via their well-known names. Clients will receive response messages on local private app-queues.

The *X♦IPC* platform environment files (the `xipc.env` files referred to as "`.env` file" in the above diagram), to be configured on servers and clients, will have the following statements:

```
[CATALOG.TCPIP]
NAMESPACE    foo:S1,S2        # Namespace "foo" anchored on S1 and S2
```

An *X♦IPC* instance will be started on all client and server nodes. These instances will all be affiliated with namespace "`foo`". This will be accomplished by coding all instance configuration files (referred to as "`.cfg` file" in the above diagram) as having the following [XIPC] sections:

```
[XIPC]
NAMESPACE    foo              # Set instance affiliation with namespace "foo"
```

5.5 Platform Configuration Parameters

Having examined platform configuration, generally, from the X•IPC catalog namespace perspective, it is now time to look more closely at the actual contents of the `xipc.env` platform configuration file. As described, X•IPC Version 3 introduces the concept of an X•IPC platform environment. This environment is the infrastructure used for supporting all X•IPC activity on that platform. Included within the X•IPC platform environment are:

An internal (hidden) X•IPC instance for supporting internal interprocess communication

A number of X•IPC daemon/service programs that operate in the background

The X•IPC catalog for supporting the X•IPC namespaces

The X•IPC catalog server program, when started by `xipcinit`, receives its configuration parameters, from within the X•IPC Environment Configuration File (i.e., `xipc.env`) that is set up on that node. The X•IPC platform environment must be configured properly in order for X•IPC-based applications running on the platform to operate properly.

The general layout of a platform configuration (`xipc.env`) file, in a TCP/IP environment, is as follows:

```
[xipcinit]
. . . any xipcinit parameters . . .

[xipclad]
. . . any xipclad parameter . . .

[xipciad]
. . . any xipciad parameters . . .

[xipcisd]
. . . any xipcisd parameters . . .

[xipcicd]
. . . any xipcicd parameters . . .

[xipcidld]
. . . any xipcidld parameters . . .

[xipcreg]
. . . any xipcreg parameters . . .

[xipcdreg]
. . . any xipcdreg parameters . . .

[CATALOG]
. . . general catalog parameters - listed below . . .

[CATALOG.TCPIP]
. . . TCP/IP specific catalog parameters - listed below . . .
```

The complete list of `xipc.env` file parameters are described in the X•IPC User Guide and Reference Manual. The following sections describe the platform environment parameters that relate to the MomSys subsystem, and in particular those parameters that deal with the X•IPC catalog server that is to run on the platform.

5.5.1 GENERAL CATALOG PARAMETERS

The table below lists the general catalog configuration parameters. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[CATALOG]` section of the configuration is not significant.

Parameter Name	Description	Default Value
[CATALOG]	Catalog section header.	- N/A -
MAX_NAMESPACES	Maximum number of *X•IPC* namespaces that can be supported within the catalog.	8
MAX_NODES	Maximum number of network nodes that can be registered within the catalog.	31
MAX_INSTANCES	Maximum number of instances that can be registered within the catalog.	31
MAX_APPQUEUES	Maximum number of app-queues that can be registered within the catalog.	128

5.5.2 PROTOCOL-SPECIFIC CATALOG PARAMETERS

The table below lists the protocol-specific catalog configuration parameters for the TCP/IP protocol. Each parameter is presented with its name, description and a default value, where relevant.

Parameter Name	Description	Default Value
[CATALOG.TCPIP]	Catalog protocol header for TCP/IP.	- N/A -
NAMESPACE	Defines an *X•IPC* namespace.	There is no default value.

5.6 Platform Utility Commands

X•IPC provides two utility commands for starting and stopping the X•IPC environment on a platform. These are the `xipcinit` and `xipcterm` utilities, respectively. These utilities refer to the platform's environment configuration file for determining the details of the platform environment being started.

5.6.1 PLATFORM STARTUP - XIPCINIT

`xipcinit` is a utility program that must be run on a platform before *any* other X•IPC work is performed on that platform. The method for invoking `xipcinit` may be platform-specific. Refer to the X•IPC Platform Notes for the respective platforms, for details.

Example:

```
# Initialize X•IPC platform environment

xipcinit
```

5.6.2 PLATFORM SHUTDOWN - XIPCTERM

`xipcterm` is a utility program that should be run on a platform when a platform is being shut down. The utility shuts down all underlying X•IPC activity occurring on the platform in an orderly manner. The syntax for invoking `xipcterm` may be platform-specific. Refer to the X•IPC Platform Notes for the respective platforms, for details.

Example:

```
# Shutdown the X•IPC platform environment

xipcterm
```

5.7 MomSys Subsystem - Instance Configuration Parameters

As we have seen, X•IPC instances play a key role in the MomSys programming model. They are the entry points for programs to access an X•IPC namespace. Furthermore, within an instance, the MomSys subsystem provides the infrastructure for supporting asynchronous store-and-forward application messaging activity.

The instance – *and in particular the MomSys subsystem* – must therefore be configured properly if the application messaging needs of a distributed application are to be met.

Note that if multiple MomSys instances are to be started on a single platform, each instance must have its configuration file in a separate directory because MomSys generates files in that directory that will conflict with one another.

This section describes the instance configuration parameters that relate to the MomSys subsystem. The parameters will be presented in the following categories, each addressing a different aspect of MomSys:

General X•IPC parameters

General MomSys parameters

Message Repository parameters

Communication Manager parameters

Protocol specific parameters

5.7.1 GENERAL X♦IPC PARAMETERS

The table below lists the general instance configuration parameters, i.e., parameters that go within the `[XIPC]` section of the instance configuration file in support of the MomSys programming model. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[XIPC]` section of the configuration is not significant. The default values shown do *not* represent limits for the values that any particular user may require.

Parameter Name	Description	Default Value
NAMESPACE	The name of the X♦IPC namespace to affiliate the instance with, or `none`, meaning that the instance is not affiliated with any namespace.	none

5.7.2 GENERAL MOMSYS PARAMETERS

The table below lists the general MomSys configuration parameters. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[MOMSYS]` section of the configuration is not significant. The default values shown do *not* represent limits for the values that any particular user may require.

Parameter Name	Description	Default Value
MAX_USERS	The maximum number of concurrent MomSys users (real users and pending asynchronous operations) that can be supported by the subsystem.	32
MAX_DISK_AQ	The maximum number of disk-based app-queues.	16
MAX_REMOTE_AQ	The maximum number of remote app-queues to be accessed at any one time.	31
MAX_MSG_LENGTH	The maximum message size. Note that when two instances are communicating, `MAX_MSG_LENGTH` must be the same for both instances; otherwise, results will be unpredictable.	1024

5.7.3 MESSAGE REPOSITORY PARAMETERS

The table below lists the message repository configuration parameters. They too are part of the `[MOMSYS]` section within an instance configuration file. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[MOMSYS]` section of the configuration is not significant.

Parameter Name	Description	Default Value
TIMEOUT_EXPIRE_MRO	The time that incomplete outbound messages are allowed to remain incomplete within the MRO. Time is specified as a string such as "12h" or "30m", etc., where the format is "nUNITS" where UNITS is: s, m, h, d or w; or `infinite` indicating that incomplete messages are never made eligible for purging.	infinite
TIMEOUT_EXPIRE_MRI	The time that inbound messages are allowed to remain undelivered within the MRI. Time is specified as a string such as "12h" or "30m", etc., where the format is "nUNITS" where UNITS is: s, m, h, d or w; or `infinite` indicating that undelivered messages are never made eligible for purging.	infinite
TIMEOUT_RETIRE_MRO	The time that "completed" outbound messages are kept within the MRO after "completing". Time is specified as a string such as "12h" or "30m", etc., where the format is "nUNITS" where UNITS is: s, m, h, d or w; or `immediate` indicating that completed messages are immediately made eligible for purging.	immediate
TIMEOUT_RETIRE_MRI	The time that delivered inbound messages are kept within the MRI after delivery. Time is specified as a string such as "12h" or "30m", etc., where the format is "nUNITS" where UNITS is: s, m, h, d or w; or `immediate` indicating that delivered messages are immediately made eligible for cleaning.	60m
SCHED_MR_CLEAN	A schedule-string defining when MomSys cleans MRI and MRO of expired or retired messages, or the string "`none`". (The syntax of a schedule-string is defined in "Scheduling Automatic MR Cleaning" later in this Guide.)	0,30 * * * * (Clean occurs every 30 minutes)
MODE_MR_CLEAN	Some combination of the following three keywords: `STARTUP` – indicating that MR clean is to occur at instance start. `SCHEDULED` – indicating that MR clean is to occur based on value of `SCHED_MR_CLEAN` `CONTINUOUS` – indicating that a partial, but incomplete, clean should occur on-the-fly.	`STARTUP SCHEDULED CONTINUOUS` (all three expressed as a single parameter, separated by spaces)
SLOT_SIZE_MRI	Should be set to the 90%-tile message size (in bytes) of messages to use MomSys. If you change the `SLOT_SIZE` in the `.cfg` file, and then attempt to restart an instance `xipcstart` may fail. You must start a fresh instance if you plan to change the `SLOT_SIZE.`	256

Parameter Name	Description	Default Value
MAX_FILES_MRI	Maximum number of disk files to be used by MRI	512
FILE_SIZE_MRI	Initial size of MRI files when created (in KBs).	`1024` (= 1 MB)
MAX_MAPPED_MEMORY_MRI	Maximum bytes of MRI data mapped into system at any one time (in KBs).	`32768` (= 32 MB)
SLOT_SIZE_MRO	Should be set to the 90%-tile message size (in bytes) of messages to use MomSys. If you change the `SLOT_SIZE` in the `.cfg` file, and then attempt to restart an instance `xipcstart` may fail. You must start a fresh instance if you plan to change the `SLOT_SIZE.`	256
MAX_FILES_MRO	Maximum number of disk files to be used by MRO	512
FILE_SIZE_MRO	Initial size of MRO files when created (in KBs).	`1024` (= 1 MB)
MAX_MAPPED_MEMORY_MRO	Maximum bytes of MRO data mapped into system at any one time (in KBs).	`32768` (= 32 MB)
DATABASE_MRI	Path of inbound message repository. Note that it is *not* possible to have MRI databases from two instances sharing a single directory; naming conflicts will occur. In such a case, set the two instances' `DATABASE_MRI` parameters to point to separate file-system directories.	Path of instance `.cfg` file.
DATABASE_MRO	Path of outbound message repository. Note that it is *not* possible to have MRO databases from two instances sharing a single directory; naming conflicts will occur. In such a case, set the two instances' `DATABASE_MRO` parameters to point to separate file-system directories.	Path of instance `.cfg` file.
JOURNAL_EXPIRED_MSGS_MRI	The fully qualified filename in which expired MRI messages are to be journaled. (See note below.)	No default
JOURNAL_RETIRED_MSGS_MRI	The fully qualified filename in which retired MRI messages are to be journaled. It may be the same as the above filename. (See note below.)	No default
JOURNAL_EXPIRED_MSGS_MRO	The fully qualified filename in which expired MRO messages are to be journaled. (See note below.)	No default
JOURNAL_RETIRED_MSGS_MRO	The fully qualified filename in which retired MRO messages are to be journaled. It may be the same as the above filename. (See note below.)	No default

NOTE: If any journal parameter is not specified, no journaling occurs for that message class. The two MRI filenames may both refer to the same file, as may the two MRO filenames, but no one file may be specified as the journal file for both MRI and MRO messages. For example, `JOURNAL_EXPIRED_MSGS_MRI` and `JOURNAL_EXPIRED_MSGS_MRO` must be distinct if they are both specified. There is no default journal file.

5.7.4 COMMUNICATION MANAGER PARAMETERS

The table below lists the communication manager configuration parameters. They too are part of the `[MOMSYS]` section within an instance configuration file. Each parameter is presented with its name, description and default value. The order that parameters appear within the `[MOMSYS]` section of the configuration is not significant. The current version supports a single communication manager parameter.

Parameter Name	Description	Default Value
MAX_INSTANCES_LINKS	Maximum number of remote instances that can be linked to this instance at any one time.	31

5.7.5 PROTOCOL SPECIFIC PARAMETERS

The table below lists the protocol-specific MomSys configuration parameters for TCP/IP. Each parameter is presented with its name, description and default value.

Parameter Name	Description	Default Value
[MOMSYS.TCPIP]	MomSys protocol header for TCPIP.	-N/A-
LINK_RETRY_INTERVAL	The time between retries of trying to create a new link. Time is specified as a string such as “12h” or “30m”, etc., where the format is “nUNITS” where UNITS is: s, m, h, d or w.	60s
LINK_PING_INTERVAL	The time between internal instance-ping messages sent to check if remote instances are still active. Time is specified as a string such as “12h” or “30m”, etc., where the format is “nUNITS” where UNITS is: s, m, h, d or w.	120s
LINK_PING_TIMEOUT	The wait time for hearing a response to an instance-ping. If no response is received, the link to the remote instance is assumed down.	60s
MSG_RESPONSE_TIMEOUT	The wait time for receiving an internal “ack” on an application message forwarded to a remote instance.	60s
QUEUE_PROBE_TIMEOUT	The wait time for a response to an internal queue-probe message. If no response is received the queue probe fails.	10s
QUEUE_PROBE_RETRY_INTERVAL	The time between queue-probe attempts.	120s

5.8 Instance Utility Commands

As detailed in the X•IPC User Guide and Reference Manual, two utilities are provided by X•IPC for starting and stopping an X•IPC instance. They are the `xipcstart` and `xipcstop` utilities respectively. X•IPC instances may also be started and stopped under program control via the XipcStart() and XipcStop() function calls. These too are described in the X•IPC User Guide and Reference Manual documentation.

The `xipcstart` and `xipcstop` utilities reference an X•IPC instance configuration file for determining the configuration details for the instance being started or stopped. The general format for instance configuration files is described in the X•IPC User Guide and Reference Manual. The following sections discuss `xipcstart` and `xipcstop` from the perspective of the MomSys subsystem.

5.8.1 INSTANCE STARTUP - XIPCSTART

Recall that in the MomSys programming model the role of the X•IPC instance is to act as an entry point for local processes to access an X•IPC namespace. Accordingly, the name assigned to the instance at create time may be flagged as being "local", since it will, typically, only be logged into by local processes.

Example:

```
# Start XIPC instance using config file /usr/harvey/applic.cfg
# Instance is assigned (local) name "abc"

xipcstart -l abc /usr/harvey/applic
```

Once the above instance has started, user programs can begin accessing it via calls to the XipcLogin() function, such as the following:

```
/*
 * Log in to instance "abc"
 */

XipcLogin ("@abc", "SomeUser");
```

5.8.1.1 Instance Recovery

`xipcstart` is instrumental in performing the recovery of an instance (i.e., its disk-based MomSys message data) following a "disorderly" system termination such as a hardware failure. When `xipcstart` executes, it recovers the state of the non-volatile MomSys subsystem to the state that it was in the last time the instance was active.

The effect of this is that if the instance being started was *not* stopped in an orderly manner during its last episode, `xipcstart` performs the necessary data recovery steps before bringing the instance up. This step can take a few moments, depending on the volume of MomSys message data resident within the instance.

5.8.1.2 Starting a Clean Instance

Occasionally, it is useful to start an instance without recovering any of the instance's prior data. In such a case, the "initialize" flag is specified as part of the `xipcstart` operation. When `xipcstart` executes, it ignores and deletes any state information about the instance's prior activity. The started instance is given a clean slate as if it never had been started and used in the past.

Refer to the description of the `xipcstart` utility and the XipcStart() API found in the X•IPC User Guide and Reference Manual for additional details.

5.8.2 INSTANCE SHUTDOWN - XIPCSTOP

`xipcstop` is employed for stopping an instance. The syntax for invoking `xipcstop` is platform-specific and is described in the X•IPC User Guide and Reference Manual.

5.9 Interactive Command Interpreter - "xipc>"

The X•IPC interactive program is an additional tool that may be used to develop, test and later support MomSys-based applications. All MomSys verbs are accessible interactively using this utility. The syntax for executing X•IPC interactive commands is defined individually per API definition in the MomSys Reference Manual.

It is thus possible to perform numerous tasks without having to write 'C' programs. Examples include:

Creating, Deleting an app-queue

Sending messages to an app-queue

Receiving messages from an app-queue

Defining MomSys events

Getting statistics on: users, app-queues, instance communication links, etc.

5.9.1 SAMPLE USAGE OF MOMSYS INTERACTIVE COMMANDS

This section presents a selection of sample sessions with the X•IPC Command Interpreter for performing MomSys operations. The examples demonstrate the types of situations where using the interactive tool can provide important time-saving development assistance. [Note: the `xipclogin` and `xipclogout` verbs are described in the X•IPC User Guide and Reference Manual.

Sample 1: Access AQid handle to app-queue "xyz" and then send a message to app-queue.

```
xipc> xipclogin @SomeLocalInstance SomeUser
      Uid = 4
      . . .
xipc> momaccess @xyz
      AQid = 1.3
      . . .
xipc> momsend 1.3 "hello world" normal shipped a wait
      RetCode = 0
      . . .
xipc> xipclogout
      RetCode = 0
```

Sample 2: Create app-queue "xyz" then receive the first message from it.

```
xipc> xipclogin @SomeLocalInstance SomeUser
      Uid = 3
      . . .
xipc> momcreate xyz
      AQid = 1.0
      . . .
xipc> momreceive 1.0 first a wait
      Text = "hello world", Length = 11
      . . .
xipc> xipclogout
      RetCode = 0
```

Refer to the X•IPC User Guide and Reference Manual for a detailed description of the interactive command processor, in general, and to the MomSys Reference Manual for MomSys-specific syntax definitions.

5.10 Monitoring MomSys Activity

As with the other *X•IPC* subsystems, MomSys supports a monitoring tool for program debugging, application monitoring and system administration.

5.10.1 MOMVIEW MONITOR AND DEBUGGER

X•IPC includes full-screen interactive monitors that provide continuous real-time views of the activities occurring within an instance's MomSys subsystem.

The `momview` utility supports the monitoring of MomSys activity within an instance. The following types of information are provided:

- Application queue data - *number of messages sent / received, activity counters*
- Users - *activity counters, blockage information details*
- Messages - *contents browsing, contents searching*
- Communication manager - *instance-link status and activity counters*

The monitoring facility does not require that applications be specially prepared for monitoring (e.g. "debug" mode). The facility can be invoked for any active *X•IPC* instance, including those of production systems out in the field, without any extra provisions, and without incurring performance overhead in the application when monitoring is *not* in use.

5.10.2 STARTING MOMVIEW

`momview` takes the following (optional) arguments, in any sequence :

- *The initial "interval" snapshot setting:* This argument defines, in milliseconds, the initial update frequency of the monitor. The default value is 1000 milliseconds.
- *The instance name to be monitored:* The default value is the string value of the "XIPC" environment.

Example:

```
momview 2000 @SomeInstance
```

The above command starts the `momview` monitor for the "@SomeInstance" instance. The initial update interval is set to 2000 milliseconds.

5.10.3 MOMVIEW LAYOUT

`momview`'s main display is matrix-like in appearance. Users logged into the subsystem and existing MomSys app-queues form the axes of the matrix. Interaction between users and app-queues is displayed in the body of the "interaction matrix."

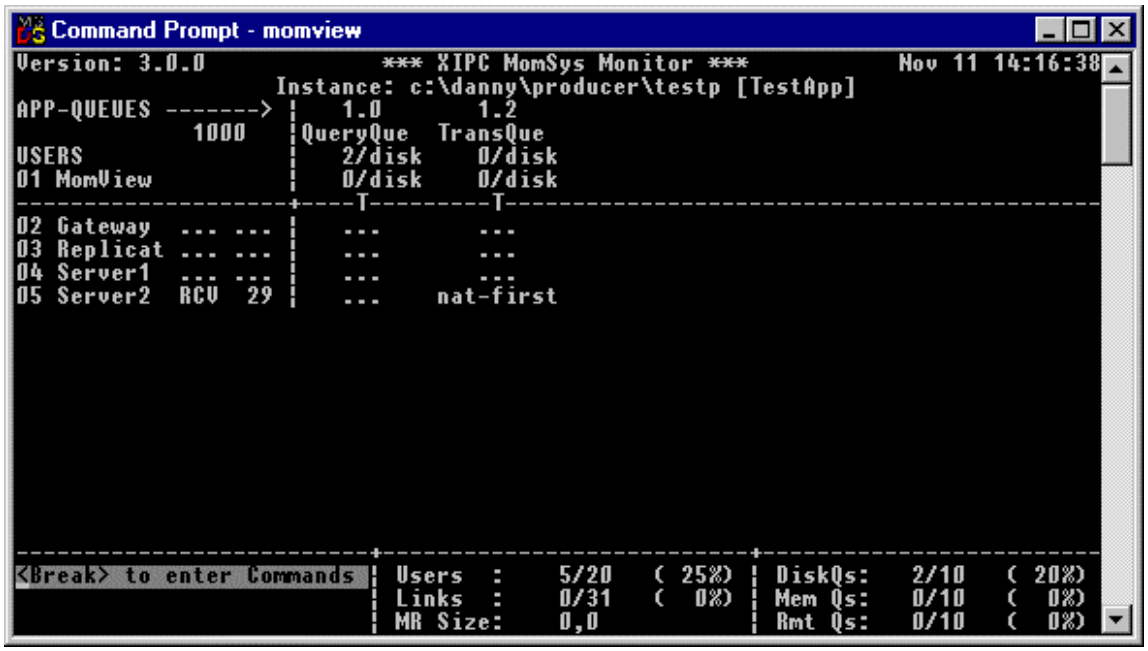
MomSys operations that block asynchronously are treated as pseudo-users of MomSys. These *Asynchronous Users* are displayed in the same manner as ordinary users, thus providing a consistent visual display of all pending MomSys asynchronous operations.

The following schematic diagram describes the various regions of `momview`'s main display window:

<table>
<tr><td>Status
Interval</td><td colspan="2">App-Queues...</td></tr>
<tr><td>Users
...
...
...</td><td colspan="2">User - Queue
Interaction
Matrix</td></tr>
<tr><td>Command</td><td>Statistics</td><td>Capacity</td></tr>
</table>

Monitor status and interval setting is shown at the top left portion of the screen. MomSys statistics and capacity data is displayed at the lower portion of the screen. The command entry window is at the lower left of the screen.

The following is a snapshot of a typical `momview` display:



Notice that the registered name of the instance being monitored is "`TestApp`". Note, as well, that this instance is based on the "`c:\danny\producer\testp.cfg`" Instance Configuration File. There are two app-queues within the instance; they are "`QueryQue`" and "`TransQue`". There are also four user currently logged into the instance. User and app-queue display elements will be described below.

5.10.3.1 User Entries

Users logged into the instance are listed on the left side of the interaction matrix, one line per user.

Each user entry includes:

- The user's MomSys user ID.
- The user's login name.
- The user's blocking status (if any).
- The blocking timeout value (if any).

In the above example, notice that User 5, logged in as "`Server2`," is blocked on a MomReceive() operation and has a timeout pending. 29 seconds remain until the operation times out.

5.10.3.2 App-Queue Entries

The instance's app-queues are identified across the top of the interaction matrix.

Each app-queue entry includes:

- The AQid of the queue. [Note that an AQid is presented as a two-integer value, e.g., `1.0`. The second integer (the 0) is an index into a table of app-queues. The first integer (the `1`) is the episode that this index has been used. Thus, after deleting AQid `1.0`, and creating another app-queue that is given index 0, the AQid of that newly created app-queue will be `2.0`.]
- The user-assigned name of the queue.
- The app-queue's message count.
- The app-queue's byte count.

5.10.3.3 Interaction Matrix Cells

Each cell on the `momview` interaction matrix describes the current relationship between an instance user and an app-queue. In the above example, notice that the intersection cell between "`Server2`" and "`TransQue`", has "`nat-first`" displayed, indicating that the user is waiting to receive the first message from the app-queue's natural sequencing of messages.

5.10.4 MOMVIEW ZOOM WINDOWS

`momview` provide the user with a variety of *zoom windows* for acquiring extended information about some aspect of the MomSys subsystem..

5.10.4.1 Zooming in on a User

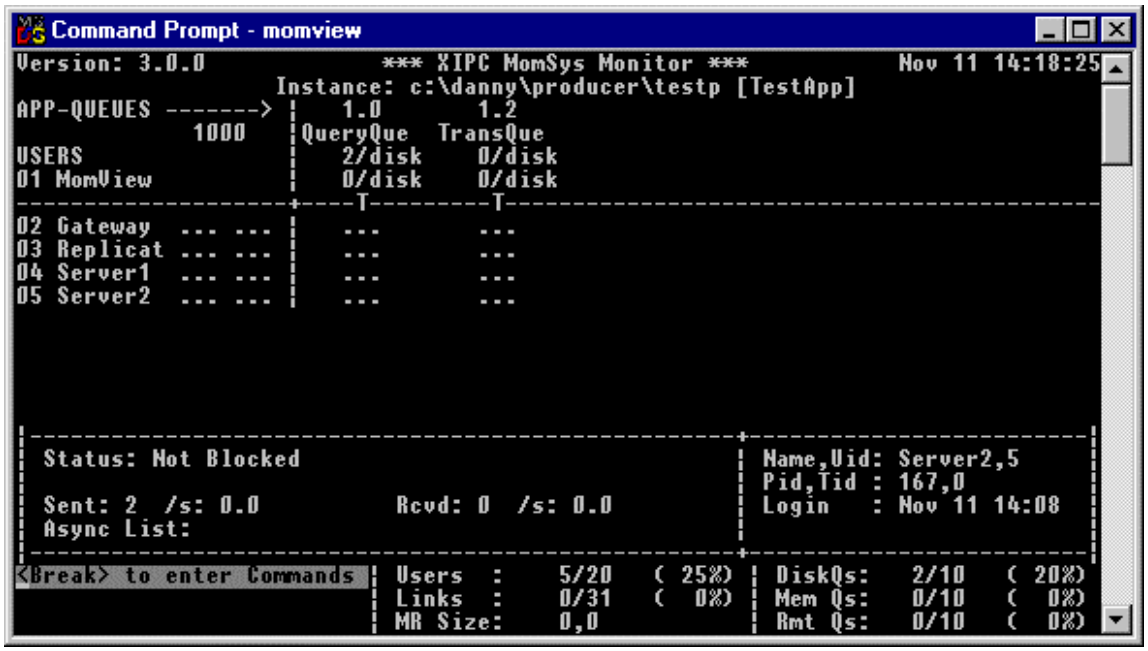
The `momview` user zoom window creates a detailed display of the status of a particular MomSys user. The command string for user zooming is "`zuN`" where N is the Uid to be zoomed on.

Example:

The command for opening a zoom window on the user having Uid of 5 is:

```
Command> zu5
```

The following display is produced:



A zoom-window is opened on User "`Server2`". In it, on the left side of the window, we see that the user is: currently "`Not Blocked`"; has sent two messages and received none since logging in; and has no outstanding asynchronous operations pending. On the right side we see more static information about the user.

Notice as well the "/s" (i.e., "per-second") values provided for sent and received messages. These values track the rate that messages are sent and received by the zoomed user.

5.10.4.2 Zooming in on a Queue

The queue zoom window provides a complete report of a queue's current status. The command string for zooming on a queue is "zqN" where N is the Qid to be zoomed on.

Example:

The command for opening a zoom window on message app-queue 1.2 is:

Command> `zq1.2`

5.10.4.3 Zooming in on the Message Repository

A zoom window is provided for providing a complete report of an instance's message repository status. The command string for the message repository zoom-window is:

Command> `zmr`

5.10.4.4 Zooming in on Instance Links

The link zoom window provides a complete report of a particular instance-link. The command string for zooming on an instance-link is "`zl`*N*" where *N* is the Link-Id to be zoomed on.

Example:

The command for opening a spool zoom window on message queue 6 is:

```
Command> zl6
```

Refer to the "Instance Links" window section below for a description of the full-screen instance-links window.

5.10.4.5 Zooming in on MomSys Subsystem Status

A zoom window is provided for providing a complete report of an instance's general status. The command string for the subsystem zoom-window is:

```
Command> zs
```

5.10.5 GENERAL MOMVIEW COMMANDS

The following are the general commands that are supported from the main `momview` window:

i*n*	Set time interval to *n* milliseconds. Example: `i100`.
zu*n*	Zoom in on user *n*. Example: `zu5`.
zq*n*	Zoom in on app-queue *n*. Example: `zq1.2`.
zl*n*	Zoom in on instance-link *n*. Example: `zl.3`
zs	Zoom in on general subsystem status information. Example: `zs`
zmr	Zoom in on Message Repository (MR) status information. Example: `zmr`
zmri	Zoom in on detailed Message Repository input (MRI) information. Example: `zmri`
zmro	Zoom in on detailed Message Repository output (MRO) information. Examp le: `zmro`
u	Un-zoom, close the zoom window.
lq	View local app-queues (This is the startup mode.)
rq	View remote app-queues
pu*n*	Pan view to user *n*. Example: `pu3`
pq*n*	Pan view to queue *n*. Example: `pq1.4`
po	Pan view to "origin", (i.e., first app-queue, first user)
l	Open the "Links" window to view the status of instance-links. Refer to the Links Window Commands below for the list of commands that can be performed from within the Links window.
b*n*	Open the "Browse" window to browse the contents messages on queue *n*, following the natural sequence. (Example: `b2.4` opens the browse window on app-queue 2.4). Refer to the Browse Window Commands below for the list of commands that can be performed from within the Browse window.
q	Quit. Exit the monitor

5.10.6 BROWSING MESSAGES WITH MOMVIEW

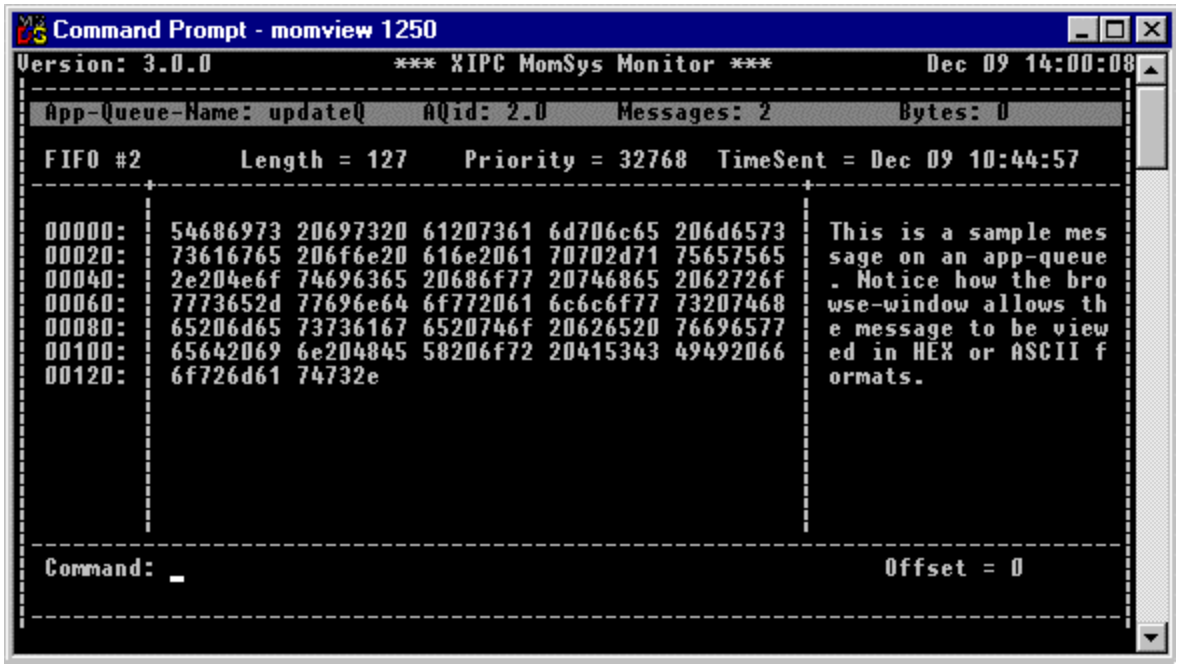
Queue and message browsing is an important feature of *X•IPC* MomSys. Using this capability, a programmer can verify a message's format or search for specific Hex or ASCII message patterns. The browse facility uses a full screen window for displaying message data. Browsing is initiated using the command string "`b`*N*", where *N* is the AQid to be browsed.

Example:

The command to initiate browsing of AQid `2.0` within an instance is:

```
Command> b2.0
```

The following is a sample display from the browse window:



The top line identifies the app-queue being browsed, in the above example "`updateQ`". The next line identifies the message within the app-queue currently being viewed. In the above example, the second message on the natural (`FIFO #2`) sequence is being shown. The message is 127 bytes in length, has a priority of 32768 (this is `NORMAL` *X•IPC* priority), and was sent Dec. 9 at 10:44:57. Note that momview can browse messages up to 10k bytes.

The body of the screen presents the message text in hex and ASCII formats. Offsets into the message are posted along the left margin.

5.10.6.1 Browse Facility Commands

Navigating in and about app-queues and individual messages is accomplished using the browse facility commands.

5.10.6.1.1 MOVING AROUND ON AN APP-QUEUE

Moving from message to message on a given app-queue can be done in a variety of ways:

Command	Effect
Þ (*right arrow*)	Move to the next message on the current sequence.
Ü *(left arrow)*	Move to the previous message on the current sequence.
n	Move to the nth message on the current sequence
+n	Move forward n messages.
-n	Move backward n messages.
f	Move to the first message on the current sequence.
l	Move to the last message on the current sequence.

Move commands work only where they make sense. Otherwise the command is ignored.

5.10.6.1.2 MOVING AROUND WITHIN A MESSAGE

Moving about *within* a message is accomplished using the following commands:

Command	Effect
Ý *(up arrow)*	Scroll the current message up one line.
ß *(down arrow)*	Scrolls the current message down one line.
PAGE-UP	Scroll the current message one page up.
PAGE-DOWN	Scroll the current message one page down.
HOME	Scroll the current message to its top.
END	Scroll the current message to its bottom.

Scrolling only works where it makes sense. Otherwise the command is ignored. Searching for a pattern within a message will cause the message to scroll to the offset where the pattern is found.

5.10.6.1.3 STRING PATTERN SEARCHING

Forward ASCII pattern searching is executed by specifying a pattern between two '/' characters and hitting return. Backward searches are specified using two '\' characters. Pattern searches can be kept confined within a single message (local), or they can cover all the messages in the current queue (global). Global search commands use a 'g' prefix. Local searches require no prefix.

The second bracket character is not always necessary, as will be demonstrated in the following examples. Repeat patterns are remembered. The following examples demonstrate these points:

Command	Effect
/ABC/	Search forward in the current message for the string "ABC".
//	Repeat the search.
/	Same.
ABC\	Search backwards in the current message for the string "ABC".
\\	Repeat the search.
\	Same.
g/ABC/	Search forward for "ABC" through all messages to the end of the queue.

g// Repeat the search.

g/ Same.

g\ABC Search backwards for "ABC" through all messages to the start of the queue.

g Repeat the search.

g Same.

5.10.6.1.4 HEXADECIMAL PATTERN SEARCHING

Searching for Hexadecimal patterns is very similar to ASCII pattern searching. The only differences are that the pattern specified is a Hex string, and that an 'x' is appended to the end of the search command.

Command Effect

/4f37/x Search forward for the hex pattern "4f37" within the current message.

g/4f37/x Same search, but forward through all messages on the queue.

g//x Same.

\4f37\x Searches backwards for the hex pattern "4f37" within the current message.

g\4f37\x Same search, but backwards through all messages on the queue.

g\\x Same.

5.10.6.1.5 SWITCHING TO ANOTHER APP-QUEUE

Switching to browse another app-queue is accomplished using the "b*N*" command as described above.

5.10.6.2 Exiting the Browse Facility

The browse facility is exited using the "q" command. Once browsing is terminated, the QueSys instance is unfrozen.

Example:

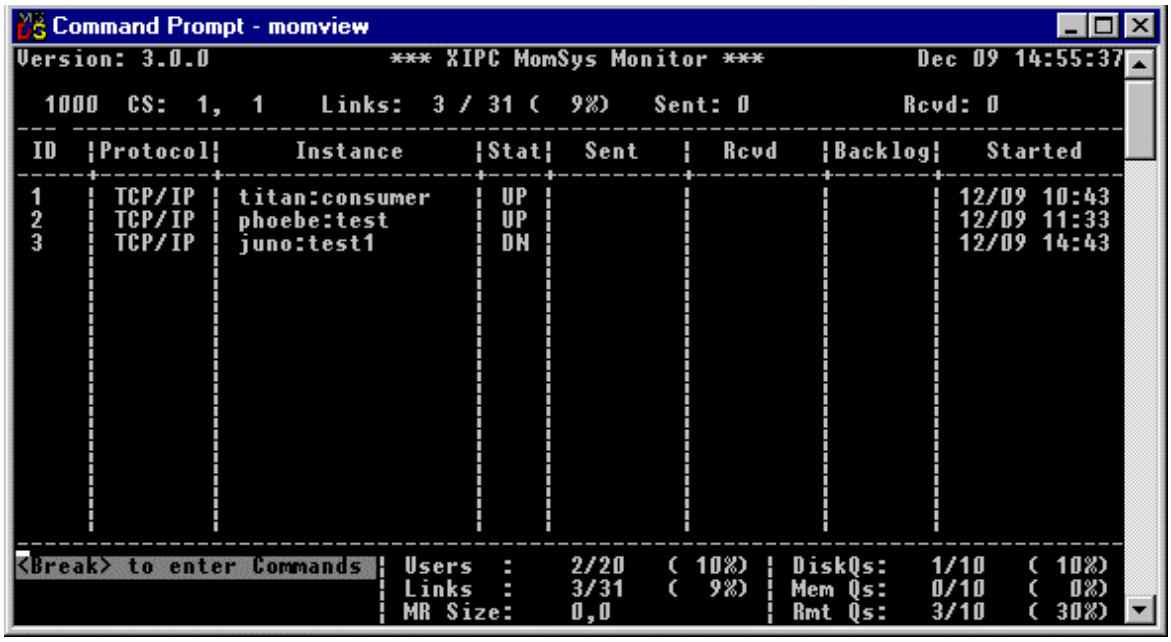
```
Command: q
```

5.10.7 MONITORING INSTANCE LINKS - THE “LINKS” WINDOW

`momview` also provides a window for monitoring the details of instance-links activity occurring within MomSys. This window is the Instance-Links (“links”) Window. The links window provides a detailed picture of all the links that exist within MomSys, including a summary of those that are active, inactive, etc.

A links window is opened using the command string `"l"`. The links window uses the top 3/4 of the monitor screen. The system statistics and command windows remain visible at the bottom of the screen.

The following is a sample display from an “instance-links” window:



Each row in the table reports another instance-link existing within the monitored instance. Various data items are provided per instance-link.

5.10.7.1 Links Window Commands

`momview` commands can be used from within the links window in the same manner that they are used from the main monitor window. Examples:

Command	Effect
i*n*	Set the interval to *n* milliseconds flow mode
b*n*	Browse the contents of app-queue *n*
q	Exit the links window

Additional commands are available that are specific to the links window. They provide a means for scrolling within the links data. These commands are:

Command	Effect
p*n*	Pan view to instance-link *n*. Example: `p5`
po	Pan view to ‘origin’, (i.e., first instance-link)

Scrolling only works where it makes sense. Otherwise, the command is ignored.

5.10.8 LOCAL AND REMOTE APP-QUEUE DISPLAY MODES

The `momview` monitor’s main window, when first brought up, presents app-queue data regarding *local* app-queues, within the monitored instance.

It is possible to change the monitor mode so that it display data regarding the *remote* app-queues known to the instance, as follows:

Example:

```
Command> rq
```

The above command causes the monitor's main window to display remote app-queue data. Returning to the local app-queue display mode is accomplished as follows:

Example:

```
Command> lq
```

5.10.9 PANNING WITHIN MOMVIEW'S MAIN WINDOW

Panning within `momview`'s main window lets the developer observe different sections of the interaction matrix. This is useful when a zoom window is open and parts of the matrix are not visible.

All "panning" commands start with '`p`'.

Vertical panning (up and down) to observe other users is done by specifying a 'u' (for user) and a Uid to pan to.

Example:

```
Command> pu8
```

The above command scrolls the interaction matrix so that Uid 8 is at the top of the display.

Horizontal panning (right and left) to monitor other queues is accomplished specifying a 'q' (for app-queue) and a AQid to pan to.

Example:

```
Command> pq1.4
```

The above command scrolls the interaction matrix so that AQid 1.4 is the first displayed (left-most).

Example:

```
Command> po
```

The command "`po`" returns the display to the origin of the activity matrix.

5.10.10 STOPPING MOMVIEW

`momview` monitoring is terminated via the 'q' command.

Example:

```
Command> q
```

Bringing down `momview` has no effect on the underlying MomSys activities. It continues to function unaffected. Any overhead incurred by monitoring is eliminated.

6. ADVANCED MOMSYS PROGRAMMING FUNCTIONALITY

6.1 Message Prioritization

The MomSend()function call defines a means for assigning prioritization to the message being dispatched relative to other messages in the system. The *Priority* argument to MomSend() is a relative value. It provides a means for indicating what urgency should be assigned a given message *as the message progresses through the system,* relative to other messages also moving within the system. The term "*as the message progresses through the system*" is a general statement that actually can be seen as having two discrete phases: The trip *to* the targeted app-queue; and the trip *through* the targeted app-queue.

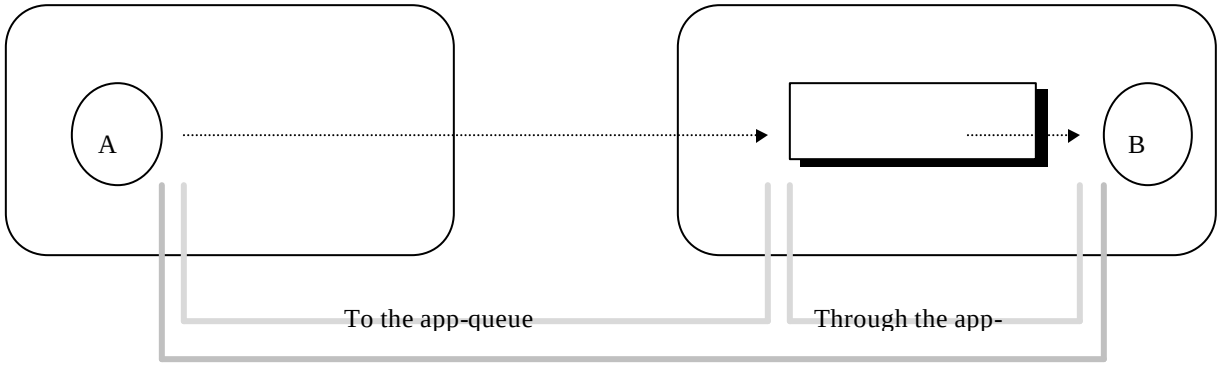
6.1.1 TWO STEPS IN A MESSAGE'S TRIP

From a prioritization perspective, MomSys messages complete their assigned trip in two steps. They are:

The trip to the targeted app-queue

The trip through the targeted app-queue.

This can be visualized as in the following diagram where a message is sent from process A to process B.



The Trip of a Message

6.1.1.1 The Trip To the Targeted App-Queue

The first phase of a MomSys message's movement to its targeted app-queue is referred to as "the trip *to* the app-queue." As depicted in the above diagram, this phase starts from the point that the message is handed off to X•IPC MomSys (via a call to MomSend()) and continues until the message has been safely placed on the targeted app-queue. During this phase a message is continuously pushed forward towards its target.

6.1.1.2 The Trip Through the Targeted App-Queue

The second phase of a MomSys message's movement is referred to as "the trip *through* the app-queue." As shown in the above diagram, this phase starts from the point that the message has been safely placed on the targeted app-queue and continues until the message is received from the app-queue.

During this phase, messages are competing with other messages on the app-queue. Thus, messages with higher priorities are pushed to the front of the app-queue's priority sequencing.

As we will see shortly, X•IPC provides the semantics for specifying a message's priority as a single value that is applied to both legs of its trip, or as two discrete values for fine-tuning each leg separately.

6.1.2 SPECIFYING MESSAGE PRIORITY VALUES

X•IPC MomSys employs a continuous scale of integers for expressing valid priority values. The lowest possible priority value is 1. The highest possible value is 65,535. The mid-way value, 32,767, is considered a "normal" priority. ***X•IPC*** provides predefined definitions for these values. They are:

MOM_PRIORITY_LOWEST	1
MOM_PRIORITY_NORMAL	32767
MOM_PRIORITY_HIGHEST	65535

In fact, a priority value may be expressed as *any* integer between 1 and 65535. *X•IPC* views them relative to one another: the higher the value, the greater the urgency.

Consider the following example:

```
/*
 * Send a message having a NORMAL priority for its entire trip.
 * The NORMAL priority value will apply to both legs of the trip.
 */

MomSend ( . . ., MOM_PRIORITY_NORMAL, . . .);
```

One could have similarly called MomSend() as follows to send a message having a slightly higher priority:

```
/*
 * Send a message having a slightly higher than NORMAL priority for its entire
 * trip. The priority value will apply to both legs of the trip.
 */

MomSend ( . . ., MOM_PRIORITY_NORMAL + 1, . . .);
```

In the next example, we will send a message that will have a high priority for getting through the system *to* the target app-queue, but it will be assigned a normal priority relative to other messages once *on* the app-queue.

```
/*
 * Send a message having HIGH priority for the trip to the app-queue.
 * Message priority through the app-queue should be NORMAL.
 */

MomSend ( . . ., MOM_PRIORITY(   MOM_PRIORITY_HIGHEST,        /* Trip to app-queue */
                                 MOM_PRIORITY_NORMAL          /* Once on app-queue */
                               ),
        );
```

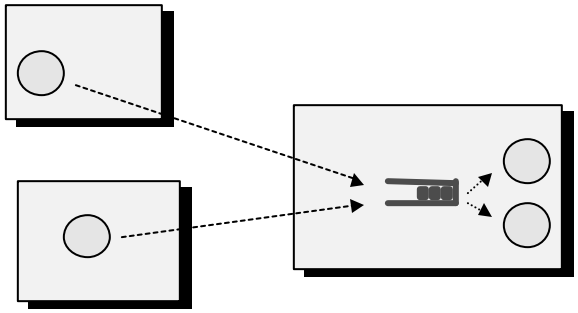
The ability to express two separate priority values for each leg of a message's movement is provided by the `MOM_PRIORITY()` macro. This macro is specified as the priority argument to MomSend(). The macro takes two valid priority values as its two arguments. The first value is the "trip *to* the app-queue priority" while the second value is the "trip *through* the app-queue priority."

Refer to Appendix B "Message Priority Specification" for additional information about this topic.

6.2 Application Message Load Management

6.2.1 LOAD SHARING

The MomSys programming model makes it natural for multiple server programs to serve requests arriving on a common application-queue.



By building the server programs so that they receive the next request off the common app-queue, it is quite easy to build application server architectures that can scale to handle a wide-range of traffic loads without having to make any special coding provisions in either the server or client programs.

6.3 MomSys Events

MomSys provides a means of monitoring the subsystem and notifying the application when certain user- defined events occur. This allows the application to take the appropriate action to handle the event.

6.3.1 THE MomEvent() FUNCTION

The MomEvent() function takes two arguments:

- *EventDescr* Description of MomSys event to be monitored
- *Notification* Notification option for announcing the occurrence of the event

These are now described.

6.3.2 SUPPORTED MOMSYS EVENTS

The following table defines the events that can be tracked. (The current version supports one MomSys event.)

MOM_EV_MSG_STATUS(*MsgId, Status)*	This event occurs when a previously sent message, identified by *MsgId*, attains a message status of *Status.* Refer to the MomSys User Guide, Appendix A, Message Status and Tracking Levels, for details on the various stages of a message's movement from sender to receiver process.
MOM_EV_APPQUE_MSGS_HI(*Aqid, NumMsgs*)	This event occurs when the number of messages on a local appqueue is greater than *NumMsgs.*
MOM_EV_APPQUE_MSGS_LOW(*AQid, NumMsgs*)	This event occurs when the number of messages on a local appqueue is less than *NumMsgs.*

6.3.3 MomEvent() "NOTIFICATION" OPTION

The following table briefly lists the possible notification options that may be specified when calling MomEvent(). More detailed descriptions may be found in the MomEvent() manual page in the MomSys Reference Manual.

MOM_WAIT	The calling process blocks synchronously until the event occurs.
MOM_TIMEOUT(*t*)	The calling process blocks synchronously for up to `t` seconds until the event occurs.
MOM_NOWAIT	The calling process synchronously checks the status of the event and returns immediately. This Notification argument can be used for polling an event, where such an approach is appropriate.
MOM_CALLBACK(*Func, &Acb*)	MomEvent() returns immediately after registering the event. The specified callback function is invoked when the specified event occurs.
MOM_POST(*Sid, &Acb*)	MomEvent() returns immediately after registering the event. The X•IPC semaphore identified by ***`Sid`*** is set when the event occurs.
MOM_IGNORE(&*Acb*)	MomEvent() returns immediately after registering the event. The ***`Acb's`*** completion flag is set when the event occurs.
MOM_SPAWN(*Command, &Acb*)	MomEvent() returns immediately after registering the event. The program specified by the ***`Command`*** string is started when the event occurs, and the *Acb's* completion flag is set as well. (Note that the ***`Acb`*** pointer can be `NULL`.) ***`Command`*** must be the path of an executable program. The ***`Command`*** string should not include any command arguments. It is not possible to pass parameters to the started command.

Consider the following example:

```
/*
 * Set event to automatically start program "HeGotIt" when a sent message achieves
 * the status of MOM_STATUS_DELIVERED.
 */

MOM_MSGID  RetMsgId;
ASYNCRESULT Acb;

RetCode = MomSend( SomeAQid,
                   "hello world",
                   12L,
                   MOM_PRIORITY_NORMAL,
                   MOM_TRACK_DELIVERED,
                   MOM_REPLY_NONE,
                   &RetMsgId,
                   MOM_WAIT);

RetCode = MomEvent(
                   MOM_EV_MSG_STATUS(   RetMsgId,
                                        MOM_TRACK_DELIVERED),    /* Set msg event */
                   MOM_SPAWN("HeGotIt",&Acb));                   /* Program to spawn */
```

An optional `MOM_RETURN` flag may be specified as part of the MomEvent() call. This is done by ORing it to the operation's *Notification* argument, as in the following example:

```
MomEvent(. . ., MOM_RETURN | MOM_CALLBACK(UserCallBack, &UserAcb,));
```

The MOM_RETURN flag (which is only valid when accompanying one of the three *asynchronous* blocking options, MOM_CALLBACK, MOM_POST or MOM_IGNORE) directs *X•IPC* to complete the operation *synchronously* if there is no need to block, and to "go asynchronous" only if the operation cannot be completed immediately. This flag allows a user to issue a MomEvent() call for creating an asynchronous event handler *only* if the event state has not occurred. Otherwise the function call returns synchronously with a return code of 0 indicating that the event state has occurred.

Events created by MomEvent() are by default *attached* to the creating *X•IPC* user. This means that, when the user logs out, the event is deleted from the system. The user may override this by logically ORing the MOM_EV_DETACHED flag to the left of the *Notification* option, in which case the event is *not* associated with the creating user.

In such a case, the user may create the event and then log out and even terminate, and still, when the event occurs, the requested action will take place.

Example:

```
/*
 * Set event to automatically start program "HeGotIt" when a sent message achieves
 * the status of MOM_STATUS_DELIVERED.
 *
 * MOM_EV_DETACHED flag allows user to log out and exit after sending the message
 * and creating the event.
 */


ASYNCRESULT Acb
MOM_MSGID  RetMsgId;

RetCode = MomSend( SomeAQid,
                   "hello world",
                   12L,
                   MOM_PRIORITY_NORMAL,
                   MOM_TRACK_DELIVERED,
                   MOM_REPLY_NONE,
                   &RetMsgId,
                   MOM_WAIT);

RetCode = MomEvent(
               MOM_EV_MSG_STATUS(   RetMsgId,
                                    MOM_TRACK_DELIVERED),        /* Set msg event */
               MOM_EV_DETACHED | MOM_SPAWN("HeGotIt",&Acb));     /* Detach from caller */
                                                                 /* Specify pgm to spawn */

RetCode = XipcLogout();

exit();
```

By specifying the MOM_EV_DETACHED option, the user is able to log out and exit after creating the event, without the event being deleted. This option is only applicable when it is ORed to the left of a *Notification* option that does *not* require the caller's continued presence. Accordingly, the MOM_EV_DETACHED flag is only valid with the MOM_SPAWN *Notification* option since it deals with an executable program which may exist independently of its creator process.

By contrast, it would be an error to specify MOM_EV_DETACHED with any other *Notification* option. This is because it doesn't make sense, for example, to specify a user callback function as the *Notification* option and expect it to be invoked *after* the process has terminated.

When MOM_EV_DETACHED is specified with MOM_SPAWN, the user's Acb will not be updated with completion information, even if the event occurs while the user is still logged in. Upon successful return from MomEvent(), the Acb will show an *AsyncStatus* of XIPC_ASYNC_DETACHED and the AUid of the associated event; the status

will never show up as "completed." The only notification that the event occurred will be the execution of the program specified to `MOM_SPAWN`.

6.3.4 MomEvent() EVENT SEMANTICS

A MomSys event is defined to have occurred whenever the monitored entity is in the awaited state. Depending on the *Notification* argument specified, MomEvent() can be used to poll the *current* state of a MomSys entity, or to wait (synchronously or asynchronously) for the entity to enter a *future* state.

When MomEvent() is called to create a new event, and the specified target is *already* in the awaited state, MomEvent() considers the event to have just occurred and the prescribed notification happens, with the qualification that the `MOM_RETURN` option can cause asynchronous notifications to return synchronously, as described above.

6.3.5 MOMSYS EVENT MONITORING

Pending MomSys events are treated as ordinary asynchronous MomSys operations in that they are assigned an Asynchronous User ID (AUid) while they are pending. The major advantage of this is that all pending asynchronous MomEvent() operations may be monitored via MomInfoUser() function calls or on the `momview` monitor. Similarly, a MomSys event may be removed from the subsystem via a call to the MomAbortAsync() function. Descriptions on employing MomSys information verbs follows in the next section.

6.4 Information Verbs

MomSys provides a number of verbs that allow a user to extract information regarding MomSys activity within an instance. The major information verbs are:

- MomInfoSys() - *Provides general, message repository and communication manager information*
- MomInfoAppQueue() - *Provides application queue information*
- MomInfoUser() - *Provides user information; also used for providing information about pending asynchronous operations and MomSys events*
- MomInfoMessage() - *Provides the latest information regarding a message*
- MomInfoLink() - *Provides information about links to other X•IPC instances*

Other secondary information verbs are provided as well for reporting less significant information occurring within the MomSys subsystem.

Using these verbs it is possible to build customized monitor processes within an application that oversee the internal operations of the application. It is additionally possible to build customized GUI-based application monitors that display data retrieved from these functions in a customized display format.

6.4.1 UNDERSTANDING MOMSYS INFORMATION VERBS

Within the *MomInfo* family of verbs there are two groups that can be employed to obtain information about a *series* of MomSys data items. The first group -- the MomInfo*Xxx*() verbs -- consists of the verbs MomInfoAppQueue(), MomInfoUser(), MomInfoLink(), and MomInfoMessage().

The programming method for looping through the series of items in this group is:

Initially, call MomInfo*Xxx*(MOM_INFO_FIRST, &...) .

Subsequently call MomInfo*Xxx*(MOM_INFO_NEXT(...), &...) .

Stop when the return code is `MOM_ER_NOMORE` .

The second group -- the `MomInfo`*`XxxX`*`List()` verbs -- consists of MomInfoAppQueueWList() and MomInfoUserAlist().

The programming method looping through the series of items in this group is:

Initially, call the corresponding `MomInfo`*`Xxx`*`()` verb, and use its output parameter both to initialize a cursor variable (e.g., `MyCursor`) to the position of the first element of the *X*List, and also to obtain information about that element

Subsequently, call `MomInfo`*`XxxX`*`List(..., &MyCursor, &...).` This advances `MyCursor` to the position of the next element of the *X*List, and then obtains information about that element.

Stop when the return code is `MOM_ER_NOMORE` .

6.4.2 CODING EXAMPLES OF MOMSYS INFORMATION VERBS

The following two code templates illustrate the two styles of information-gathering loops (including error checking).

Example 1:

```
/*
 * Sample of MomInfoXxx()verb usage - e.g. for MomInfoAppQueue().
 * Loop through all the app-queues in the current instance,
 * retrieving and processing the status data of each app-queue.
 */

MOMINFOAPPQUEUE MyInfoAppQueue;
XINT            RC, MyAQid;


for (RC  = MomInfoAppQueue( MOM_INFO_FIRST, &MyInfoAppQueue );
     RC != MOM_ER_NOMORE;
     RC  = MomInfoAppQueue( MOM_INFO_NEXT( MyAQid ), &MyInfoAppQueue ))
{
    if (RC < 0)
    {
        /* Take appropriate error action for MyInfoAppQueue */
        . . .
        break;
    }

    MyAQid = MyInfoAppQueue.AQid;

    /* Process MyInfoAppQueue data for MyAQid */
    . . .

} /* for */
```

Example 2:

```
/*
 * Sample of MomInfoXxxXList() verb usage - e.g. for MomInfoAppQueueWList().
 * Loop through the entire wait-list for the specific app-queue
 * identified by MyAQid, retrieving and processing the status
 * data of each wait-list element.
 */


XINT                    RC, MyAQid, MyCursor;
MOMINFOAPPQUEUE         MyInfoAppQueue;
MOM_APPQUEUEWLISTITEM   MyWListItem;


MyAQid = ...; /* AQid of app-queue whose wait-list is to be traversed */

if ( (RC = MomInfoAppQueue( MyAqid, &MyInfoAppQueue )) < 0 )        /
{
    if (RC != MOM_ER_NOMORE)
    {
        /* Take appropriate error action for MomInfoAppQueue() */
        . . .
    }
}
else    /* we have at least one element in the wait-list */
{
    for (MyCursor       = MyInfoAppQueue.WListInitialCursor,
            MyWListItem = MyInfoAppQueue.WListFirstItem;
         RC != MOM_ER_NOMORE;
         RC = MomInfoAppQueueWList( MyAQid, &MyCursor, &MyWListItem ))

    {
        if (RC != MOM_ER_NOMORE)
        {
            /* Take appropriate error action for MomInfoAppQueueWList */
            . . .
            break;
        }

        /* Process MyWListItem data */
        . . .

    } /* for */

} /* else */
```

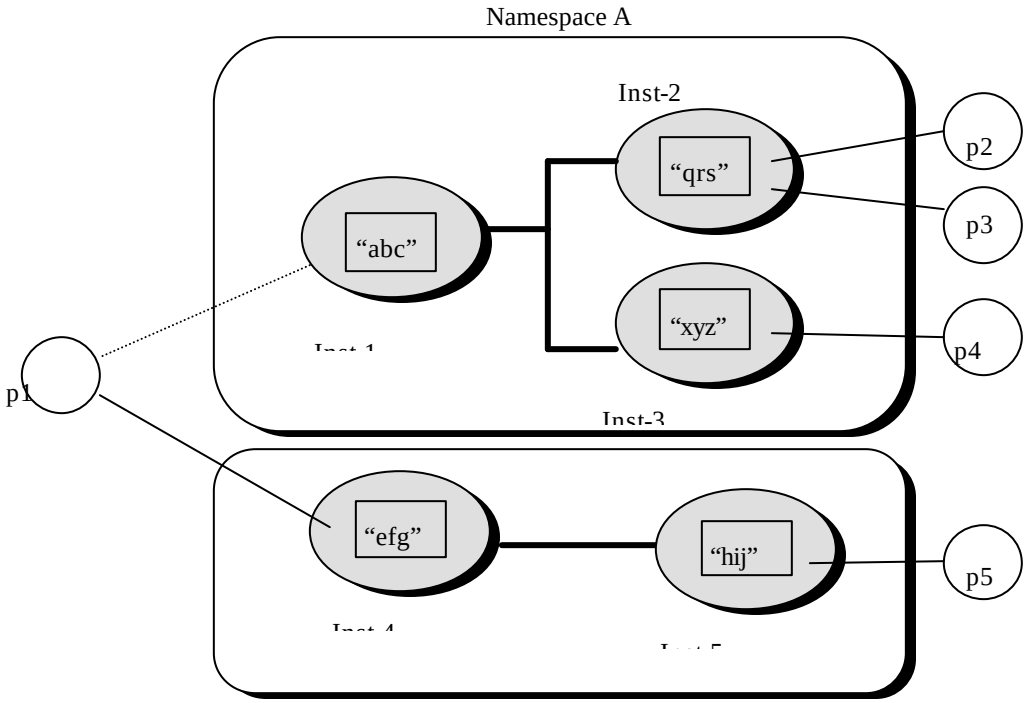
If one wanted to loop through *all* the app-queues' wait-lists, then the second code segment above would be nested in the first segment, so that the processing of each app-queue would entail traversing its wait-list.

Refer to the respective Reference Manual pages for additional details on the usage of these verbs.

7. ADVANCED MOMSYS CONFIGURATION CONCEPTS

7.1 Accessing Multiple Namespaces

As pointed out earlier, X•IPC supports the possibility of multiple namespaces being active in an environment for building applications that require such a form of partitioning. Consider the following diagram:



Namespace B

Two X•IPC namespaces are active: namespace A and namespace B. Of all the processes accessing the two namespaces, only p1 is accessing both: It access namespace A via its login to Inst-1 and it accesses namespace B via its login to Inst-4. Herein lies the approach for accessing multiple namespaces.

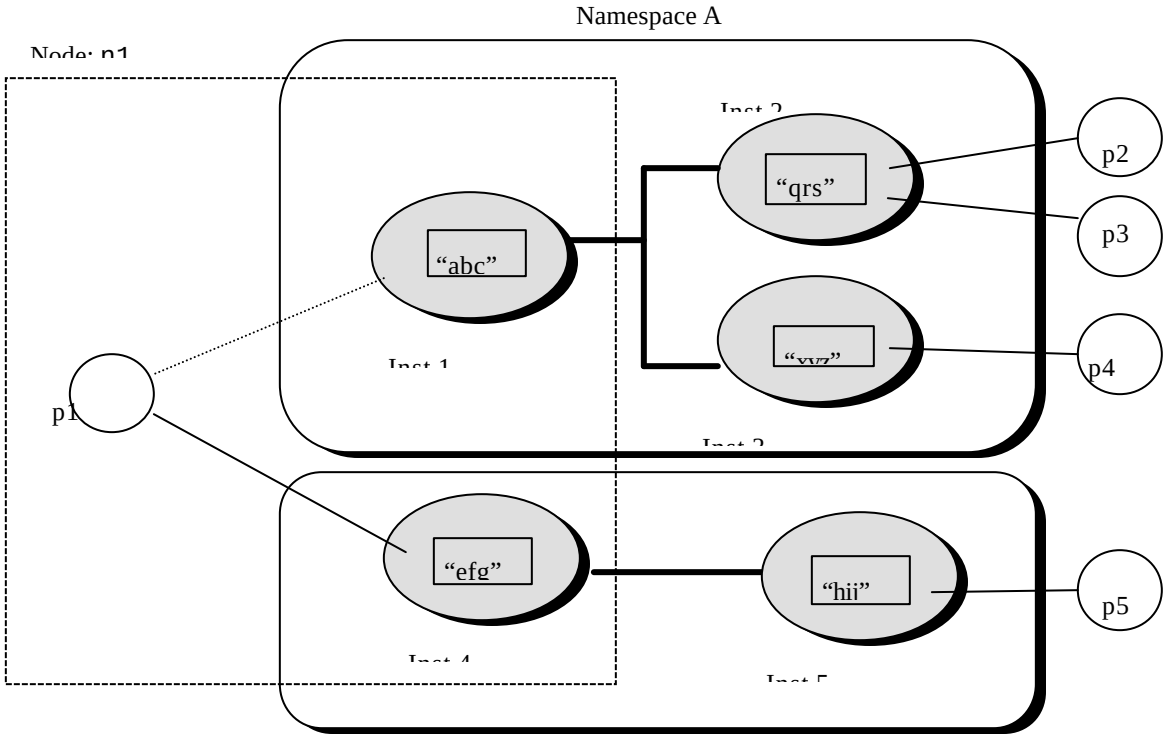
A process wishing to access multiple namespaces does so by logging into multiple instances, one per namespace. As generally is the case with X•IPC , a process may log into multiple instances, but at any point in time only one login is considered its *current* login. The toggling between logins is accomplished via the XipcConnect() and XipcDisconnect() function calls. Refer to the X•IPC User Guide and Reference Manual for discussions on how and when to use XipcConnect() and XipcDisconnect() for toggling between multiple logins.

Here too, a process such as p1 can have only one current login, either its login to Inst-1 or its login to Inst-4. In the above diagram, the solid line indicates a current login, while the broken line indicates a login which is not currently connected. Process p1's current login is its login to Inst-4.

Because p1's current login is to Inst-4 , p1's current namespace is namespace B. If and when p1 will wish to access namespace A it will need to make its login to Inst-1 current. This is accomplished via calls to XipcConnect() and XipcDisconnect().

7.2 Configuring X♦IPC 's Platform Environment for Multiple Namespaces

It is possible for a platform to support instances that are affiliated with different namespaces.. Consider the prior example:



Process p1 is accessing the two namespaces via instances Inst-1 and Inst-4, all of which reside on a single platform, as depicted by the dashed box. The X♦IPC platform environment for that platform will require that both namespaces A and B are identified within the `xipc.env` file's `NAMESPACE` statements as namespaces to which instances will become affiliated when started. .

Assuming that the two namespaces are anchored within the catalog server on `n1`, the `xipc.env` file for `n1` would contain the following statements:

```
[CATALOG.TCPIP]
NAMESPACE     A:n1
NAMESPACE     B:n1
```

Similarly, the instance configuration files for Inst-1 and Inst-4 would have the following `NAMESPACE` statements:

```
# Statement within configuration file for instance "Inst-1"

[XIPC]
NAMESPACE     A

# Statement within configuration file for instance "Inst-4"

[XIPC]
NAMESPACE     B
```

8. ADVANCED MOMSYS ADMINISTRATION CONCEPTS

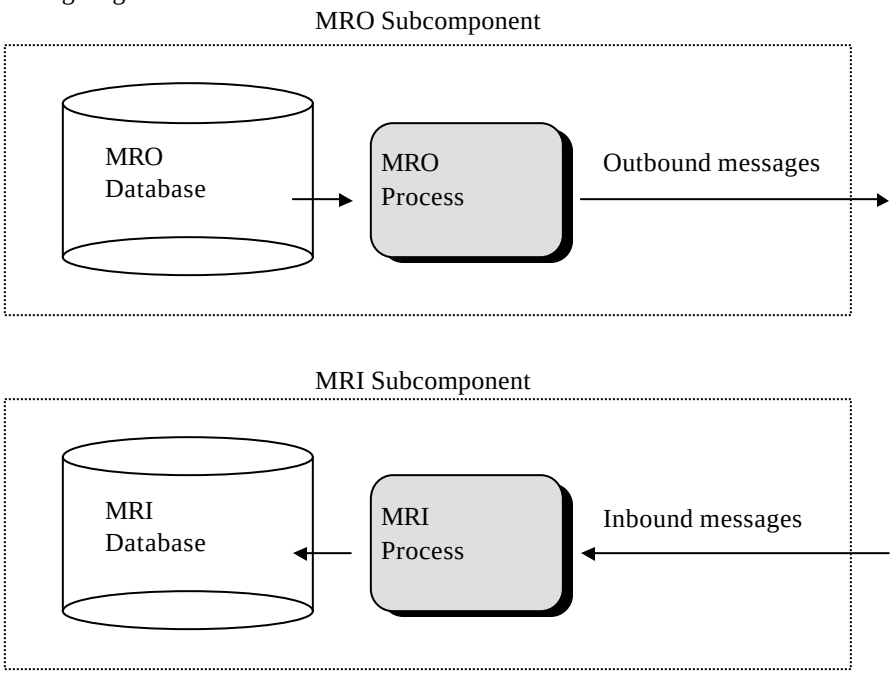
8.1 Message Repository

The MomSys message repository (MR) is one of the more complex components of the MomSys subsystem. It supports all aspects of MomSys message manipulation that are non-volatile and recoverable. Basic utilization of MomSys does *not* require detailed knowledge of how the message repository is built and operates. Such knowledge, however, can become useful to users who are interested in achieving optimizations and performing advanced operations within the message repository. This section introduces some of these advanced concepts.

8.1.1 COMPONENTS

We saw above that there is one message repository per X•IPC instance. While conceptually correct, this is not technically accurate. An instance's message repository is in fact divided into two parts: an MRO (Message Repository Outbound) subcomponent, and an MRI (Message Repository Inbound) subcomponent. These are now described.

Consider the following diagram:



An instance's message repository is comprised of an MRI and an MRO subcomponent. These subcomponents are each comprised a process and a database for supporting inbound and outbound messages, respectively. Sent messages, in the course of being moved out of the local instance to a disk-based app-queue within a remote instance, are moved through the MRO. Inbound messages, received from remote instances, are received through the MRI.

The MRI and the MRO operate asynchronously with respect to one another. This leads to enhanced performance in general (compared to a single component architecture), as well as to the potential for certain configuration optimizations.

8.1.2 OPTIMIZATION

One potential benefit of the message repository's split architecture is that it provides the potential to physically locate the two subcomponent databases on separate disks, where that is supported by the underlying hardware and operating system. Doing so enhances overall MR performance because of the inherent parallelism that is leveraged in such a configuration.

The location of the MR database files is defined within the `[MOMSYS]` section of an instance's configuration file via the `DATABASE_MRI` and `DATABASE_MRO` parameters.

While it *is* possible to have two instances coexisting in a single operating system directory, it is *not* possible to have MR databases from two instances sharing a single directory. Naming conflicts will occur. In such a case the `DATABASE_MRI` and `DATABASE_MRO` parameters of the two instances should be set to point to separate file-system directories.

8.1.3 MESSAGE EXPIRATION

Messages that are sent to an app-queue, and that do *not* reach their destination within a certain period of time have the potential to become *expired*. Expired messages are eventually removed from the MomSys message repository. When removed from the MR, an expired message may be entirely deleted (no remaining record kept), or it may be logged to a journal of expired messages. All aspects of this process are configurable by the user.

Messages by default are never expired. This means that, by default, all MomSys messages are assigned an "`infinite`" expiration timeout period. This can be overridden via a instance configuration MR expiration timeout parameters. Refer to the discussion on MomSys configuration for a review of these MR parameters.

8.1.4 MESSAGE RETIREMENT

Messages that successfully reach their objectives may, as well, become subject to eventual removal by X•IPC 's MR clean-up processing. The governing factor here is referred to as the message's *retirement* time-out period. The event that causes a message to become classified as retired is different per MRI and MRO.

An outbound message, within an MRO, starts a timed count-down to retirement as soon as it has becomes "complete". The message is "complete" when the message has achieved the tracking-level that was specified at the time that the message was sent. (Refer to MomSend() and Appendix A for details about tracking levels.)

An inbound message, within an MRI, starts a timed count-down to retirement when the message is successfully delivered to a process.

The message retirement time-out period within an MRO is by default "`immediate`." This means that, by default, all outbound MomSys messages that are "completed" are immediately retired and become candidates for cleaning. This default MRO retirement time-out period can be overridden via instance configuration parameters.

The message retirement time-out period within an MRI is by default set to 60 minutes (`60m`). This means that, by default, all inbound MomSys messages that are successfully delivered are kept for one hour before they are retired. This 60 minute-deep cache of received messages is retained by the MRI in order to detect duplicate messages that might be sent in the event of, for instance, a line crashing before the MRI can send a receipt acknowledgement back to the MRO. In such a circumstance, the MRO might resend a message that had in fact been received; this MRI cache of recently received messages ensures that a duplicate message is not delivered to the instance. (As with other instance configuration parameters, the `TIMEOUT_RETIRE_MRI` parameter can be overridden. See section 0.)

Retired messages are eventually removed from MRs within MomSys. When removed from its MR, a retired message may be entirely deleted (no remaining record kept), or it may be logged to a journal of retired messages. As with expired messages, aspects of this process are configurable by the user in the instance configuration file.

8.1.5 MR CLEANING

MR cleaning is an important administrative function in X•IPC MomSys. It is by means of this facility that the size of an MR database can be kept under control over long periods of ongoing operation.

The purpose of MR cleaning is to remove from the MR those messages that have either *expired* or *retired* since the last time MR cleaning was performed. Message expiration and retirement are terms that have precise meanings that were reviewed in the prior sections.

MR cleaning may be caused to occur either *automatically* or *manually*. Note that there is a default setting of every 30 minutes (represented as `0,30 * * * *` ; see the explanation below).

8.1.5.1 Scheduling Automatic MR Cleaning

The MR-clean processing of an *X•IPC* instance can be scheduled to occur at a designated set of times. As long as the instance is active at those time, the MR cleaning will occur automatically, and without the need to stop any supported applications. Configuring the schedule of when MR clean is to occur is done via the `SCHED_MR_CLEAN` instance configuration parameter.

The `SCHED_MR_CLEAN` configuration parameter is a string value indicating when MR cleaning will take place. The default value of this parameter is "`none`" meaning that no automatic cleaning is desired.

Where scheduled cleaning is desired a schedule-string having five discrete fields is specified as the parameter value. The five fields provide five levels of granularity over the definition of a schedule. These fields, and their basic definitions are now listed. More advanced possibilities are described below.

Minutes - Defines at what *five-minute* intervals within each hour to perform the MR clean.
Valid values are `0, 5, 10, . . ., 55`; or `*`, where `*` indicates *all* the values.

Hours - Defines at what hour intervals to run the MR clean.
Valid values are `0,1, 2, . . ., 23`; or `*`, where `*` indicates *all* values.

Month-day - Defines what day within each month to run MR clean.
Valid values are `1, 2, 3, . . ., 31`; or `*`, where `*` indicates *all* values.

Month - Defines what months to run MR clean
Valid values are `1, 2, 3, . . ., 12`; or `*`, where `*` indicates *all* values.

Week-day - Defines what days of the week to run MR clean
Valid values are `0, 1, 2, . . ., 6`; or `*`, where `0` is Sunday and `*` indicates *all* values.

Understanding this syntax is best accomplished by examples:

Example:

```
SCHED_MR_CLEAN   0 0 * * *    # defines a schedule that occurs once each day,
                              # at midnight
```

Example:

```
SCHED_MR_CLEAN   0 0 * * 1    # defines a schedule that occurs every Monday,
                              # at midnight
```

A field value is defined as a comma-delimited list of one or more *elements*. An *element* is either a valid number, or two valid numbers separated by a hyphen indicating an inclusive range. Note that the specification of days may be made in two fields (month-days and week-days). If both are specified as a list of elements, both are adhered to.

Example:

```
SCHED_MR_CLEAN 0 0 1,15 * 1  # defines a schedule that occurs on the first
                             # and fifteenth of each month, as well as on
                             # every Monday, at midnight
```

To specify only one of the *day* fields, the other *day* field should be set to `*`.

Example:

```
SCHED_MR_CLEAN   0 0 1,15 * *     # defines a schedule that occurs on the first
                                  # and fifteenth of each month, at midnight
```

8.1.5.2 The "mrclean" Utility Program

A utility program called "`mrclean`" is provided, as well, for *manually* executing the MomSys clean operation on an instance's message repository.

The `mrclean` utility takes one optional argument when executed:

InstName: The instance file name of the instance, or the registered name of the instance - prepended with an '@' - to be MR cleaned. The default value is the value of the `XIPC` environment variable.

Example:

```
# Run mrclean utility on local instance having registered name "foo".

mrclean @foo
```

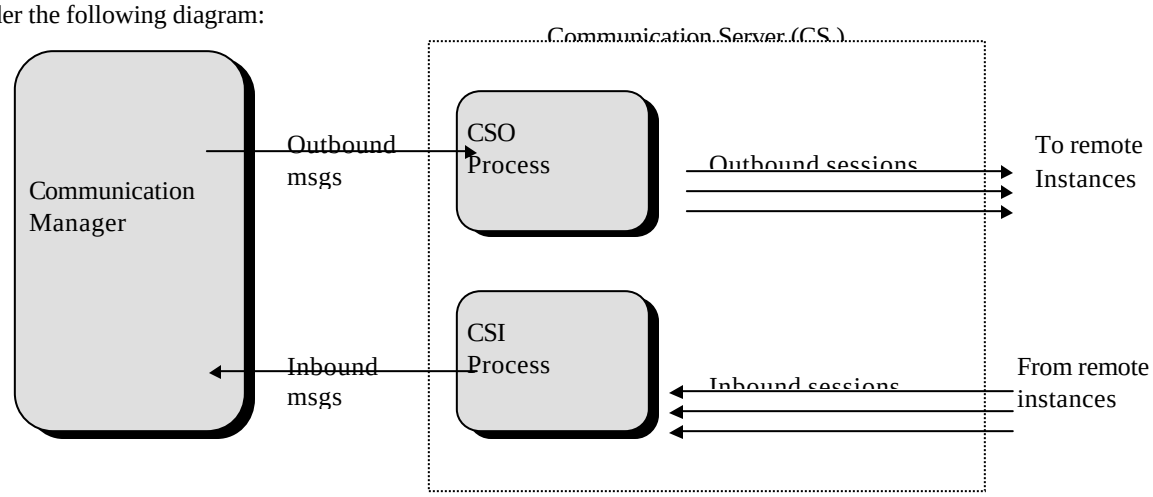
8.2 Communication Manager

An instance's communication manager (CM) supports all message communication between the instance and other instances. Basic utilization of MomSys does *not* require detailed knowledge of how the communication manager is built and operates. Such knowledge, however, can be useful to users who are interested in achieving optimizations. This section introduces some of these advanced concepts.

8.2.1 COMMUNICATION SERVERS

Actual protocol-level communication activity in an instance is handled by one or more Communication Servers (CS). Each communication server is comprised of a CSI (inbound) and a CSO (outbound) pair of subcomponents.

Consider the following diagram:

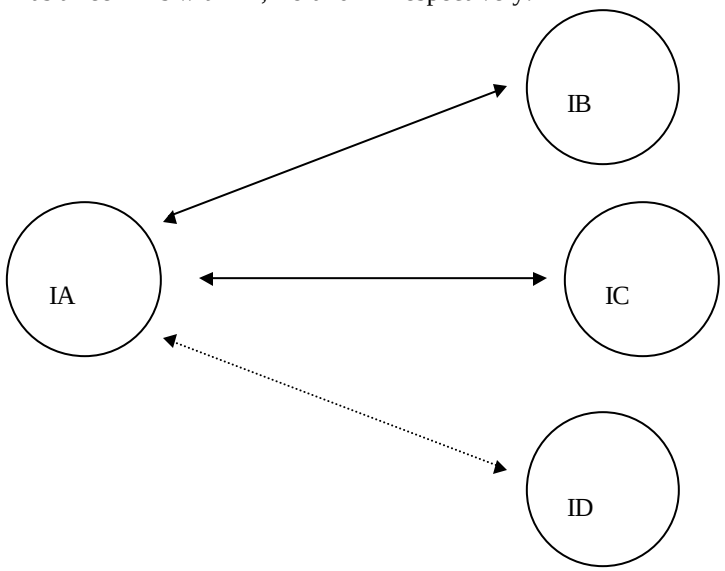


These CSI and CSO processes support inbound and outbound messages, respectively. In the course of being moved out of the current instance to a remote instance, messages are moved through the CSO. Inbound messages, received from other instances, are moved through the CSI.

CSI and CSO operate asynchronously with respect to one another. This leads to enhanced performance in general (compared to a single component architecture).

8.2.2 INSTANCE LINKS

An instance-link is defined as a two-way communication connection between two instances. In the following diagram, instance `IA` has three links with `IB`, `IC` and `ID` respectively.



Instance-links may be *up* or *down* at any point in time, depending on the current state of the underlying protocol connectivity. The links `IA-IB` and `IA-IC` are *up* in the above diagram. Link `IA-ID` is currently *down*

The current state of instance-links in an instance can be found via the `momview` monitor's "instance-links" window, or via the MomInfoLink() function call. The second approach is most useful when invoked using the *X•IPC* interactive command monitor. Refer to that function's definition in the MomSys Reference Manual for details.

Example:

```
xipc>  mominfolink first
       LinkId: [1]
       Remote Node: 'helios'      Remote Instance: 'test'
       Network Protocol: TCPIP   Link Status: DOWN
       CountMsgSent: 29880  CountMsgReceived: 102 NumBacklogMsg: 24
       StartupTime: Wed Sep 30 19:05:10 1996

xipc>  mominfolink all

       Id      Instance          Protocol Status   Messages
       ---------------------------------------------------
       1       helios:test       TCP/IP   DOWN     24
       2       titan:test1       TCP/IP   UP       14
       3       juno:product      TCP/IP   UP       0
       4       moon:test2        TCP/IP   DOWN     1040
```

X•IPC's management of links (timeout intervals, retry times, etc.) are all configurable in the instance configuration file. As was described earlier in this guide, configuration of TCP/IP parameters are set within the `[MOMSYS]` and `[MOMSYS.TCPIP]` sections of the instance configuration file

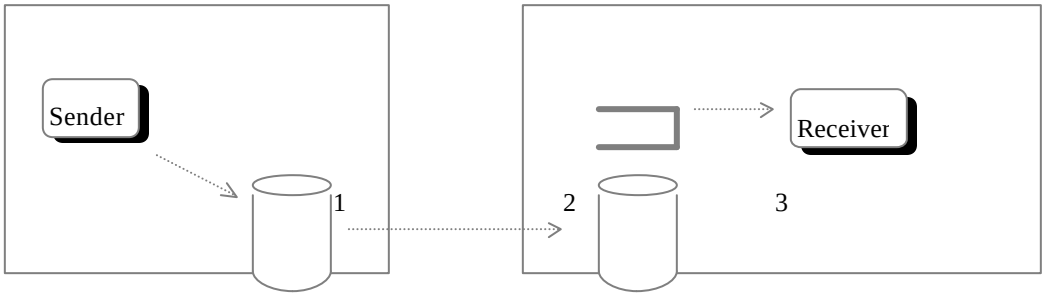
For example, it is possible to define the maximum number of remote instances accessible by an instance. This is defined via the `MAX_INSTANCE_LINKS` MomSys configuration parameter. Its default value is 31, meaning that an instance, by default, may communicate with up to 31 other instances. This may be overridden to a higher value.

Other link-related configuration, particularly those dealing with protocol-specific parameterization, are possible as well. These parameters were listed earlier in this document. Refer to the MomSys Reference Manual for additional details.

9. APPENDICES

9.1 Appendix A: Message Status and Tracking Levels

An understanding of message movement in the *X♦IPC* MomSys programming model is central to proper utilization of the subsystem. Consider the following diagram:



9.1.1 MESSAGE STATUS VALUES

An *X♦IPC* MomSys message goes through three well-defined, trackable stages as it moves from sender to receiver program. These stages are identified numerically in the above diagram. The message status values that correspond to these stages are:

MOM_STATUS_HELD	Message is currently held in the sender's local message repository, but has not yet been shipped to the receiver node.
MOM_STATUS_SHIPPED	Message has shipped to receiver's message repository and has been logically inserted in the targeted app-queue, but has not been received and removed by a receiving program.
MOM_STATUS_DELIVERED	Message has been received and removed by a receiving program.

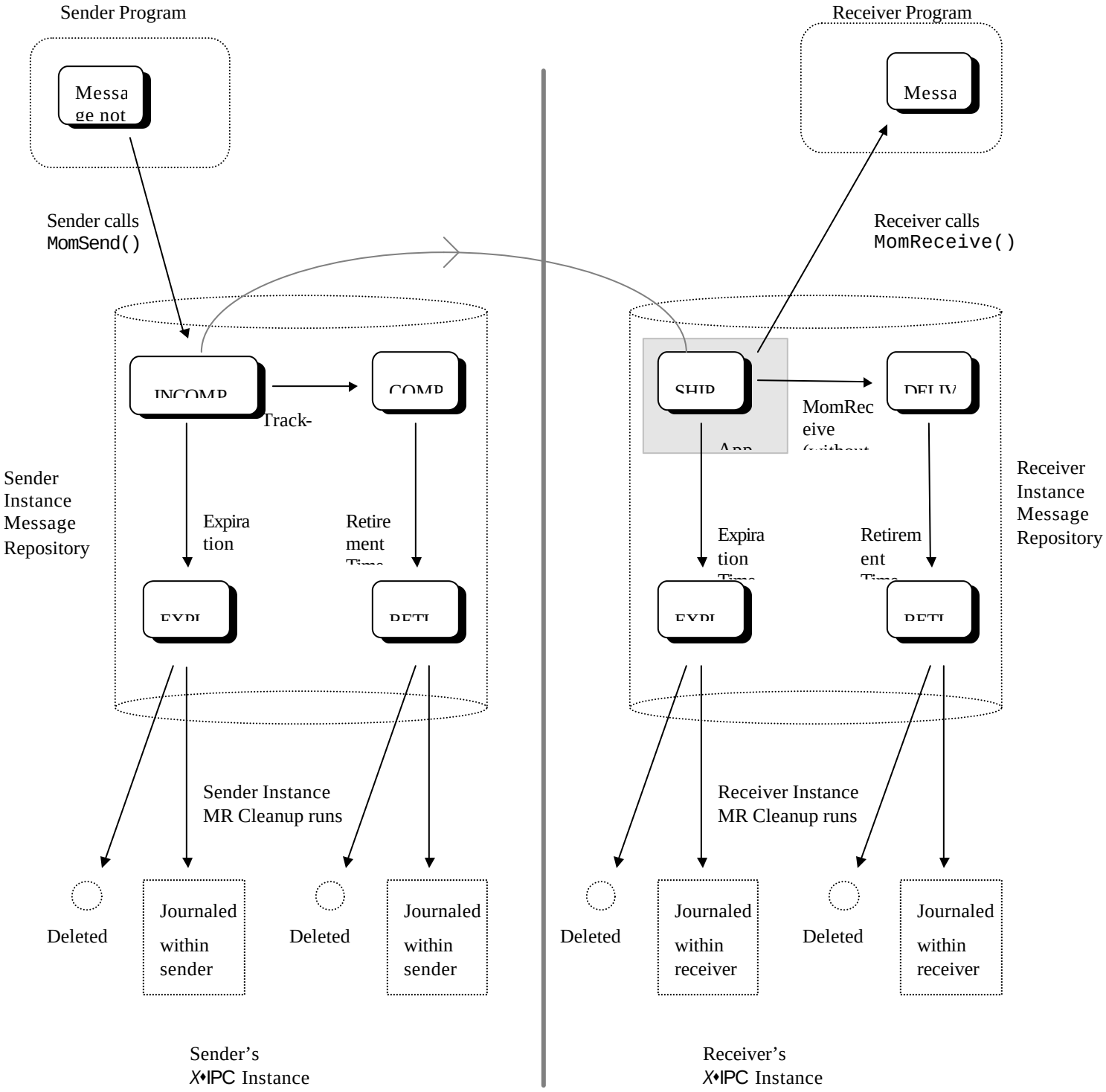
Two additional pseudo-status values that are occasionally employed in MomSys are:

MOM_STATUS_COMPLETE	Message status has achieved the tracking level that was specified for it when the message was sent via MomSend().
MOM_STATUS_INCOMPLETE	Message status has not yet achieved the tracking level that was specified for it when the message was sent via MomSend().

MomEvent() is an example of a function that employs the `MOM_STATUS_COMPLETE` pseudo-status value for creating an event that occurs when a given message reaches the tracking level that it was sent with.

Refer to the description under MomEvent() for details.

9.1.2 MESSAGE STATE-DIAGRAM



9.1.3 MESSAGE TRACKING LEVELS

Just how far a message is actually tracked by *X♦IPC* is a function of the tracking-level that is specified in the MomSend() verb when the message is sent. The two message tracking levels that may be specified are:

MOM_TRACK_SHIPPED	Track message being sent until it has attained status of `MOM_STATUS_SHIPPED`.
MOM_TRACK_DELIVERED	Track message being sent until it has attained status of `MOM_STATUS_DELIVERED`.

Note that a message status is updated in the sender's message repository up to the level requested by the tracking level argument of the MomSend() function, *but no further*. Thus, a message sent with a tracking level of `MOM_TRACK_SHIPPED` is tracked up to the point that the message attains a status of `MOM_STATUS_SHIPPED`, from which point no further tracking is performed.

9.2 Appendix B: Message Priority Specification

9.2.1 INTRODUCTION

The MomSend() function call defines a means for assigning prioritization to the message being dispatched relative to other messages in the system. This is defined in the Reference Manual pages as the *Priority* argument to the two functions.

The *Priority* argument is a relative value. It provides a means for indicating what urgency should be assigned a given message, *as the message progresses through the system,* relative to other messages, also moving within the system. The term "*as the message progresses through the system*" is a general statement that actually can be seen as having two discrete phases: The trip to the targeted app-queue and the trip through the targeted app-queue.

This Appendix describes these two phases as they relate to message prioritization. In it we will review the following topics:

The two phases in a message's journey.

Why prioritization matters.

Specifying message priority values.

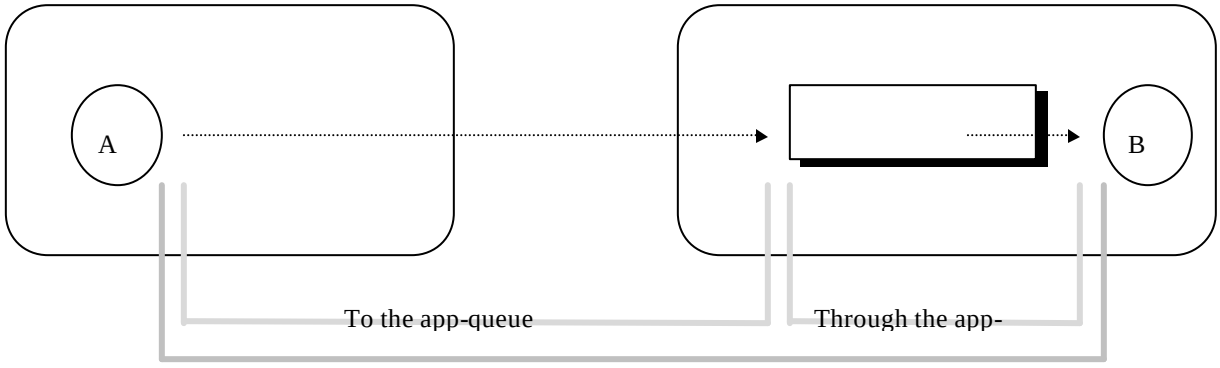
9.2.2 TWO STEPS IN A MESSAGE'S JOURNEY

From a prioritization perspective, MomSys messages complete their assigned trip in two steps. They are:

The trip to the targeted app-queue

The trip through the targeted app-queue

This can be visualized as in the following diagram where a message is sent from process A to process B.



The Trip of a Message

9.2.3 WHY PRIORITIZATION MATTERS

9.2.3.1 The Trip To the Targeted App-Queue

The first phase of a MomSys message's movement to its targeted app-queue is referred to as "the trip to the app-queue." As is depicted in the above diagram, this phase starts from the point that the message is handed off to X•IPC MomSys (via a call to MomSend()), and continues until the message has been safely placed on the targeted app-queue. During this phase a message is continuously pushed forward toward its target.

During this phase as well, a message competes with other messages in the system for attention in affecting its forward movement. The more messages flowing within the system the more the competition. It is for this reason that message prioritization is important during this phase. Suppose an application is to be written in which there will be numerous processes co-resident with the above process 'A' in the above diagram, all of which are sending messages out to the world at a busy clip. Assume further that the messages being sent are not of equal urgency. It would be useful to be allowed to assign relative levels of urgency for the messages as they push towards their targeted app-queues. We will see shortly that X•IPC provides such a mechanism.

9.2.3.2 The Trip Through the Targeted App-Queue

The second phase of a MomSys message's movement is referred to as "the trip through the app-queue." As shown in the above diagram, this phase starts from the point that the message has been safely placed on the targeted app-queue and continues until the message is received from the app-queue.

During this phase, messages are competing with other messages on the app-queue. Thus, messages with higher priorities are pushed to the front of the app-queue's priority sequencing.

This form of prioritization is useful if an application is being written in which multiple process are to send messages to a common app-queue, where it is important that certain messages be served ahead of others in the app-queue, regardless of the arrival time. It would be useful, in such a case, to be allowed to assign relative levels of urgency to the messages as they are placed on the app-queue. We will see shortly that X•IPC provides such a mechanism, as well.

9.2.3.3 Two Priority Values or One?

In many situations the urgency for the two phases of a message's trip is the same. As an example, a message may be a high-priority message. Period. In such a case the sender is interested in having the message move to the targeted app-queue *and* through the targeted app-queue as fast as possible. For such situations, X•IPC allows the user to specify a single priority value that is then applied to both phases of its movement.

As we will see shortly, X•IPC provides the semantics for specifying a message's priority as a single value that is applied to both legs of its trip, or as two discrete values for fine-tuning each leg separately.

9.2.4 SPECIFYING MESSAGE PRIORITY VALUES

9.2.4.1 Range of Priority Values

X•IPC MomSys employs a continuous scale of integers for expressing valid priority values. The lowest possible priority value is 1. The highest possible value is 65,535. The mid-way value 32,767 is considered a "normal" priority.

X•IPC provides predefined definitions for these values. They are:

MOM_PRIORITY_LOWEST	1
MOM_PRIORITY_NORMAL	32767
MOM_PRIORITY_HIGHEST	65535

In fact, a priority value may be expressed as *any* integer between 1 and 65,535. X•IPC views them relative to one another: the higher the value, the greater the urgency.

9.2.4.2 Semantics for Expressing Priorities

As described earlier, X•IPC MomSys supports two forms of message prioritization: a single priority that is assigned to both phases of a message's trip, or a pair of priority values, one for governing the trip to the app-queue and one for the trip through the app-queue. We will now examine examples of both.

Consider the following example:

```
/*
 * Send a message having a NORMAL priority for its entire trip.
 * The NORMAL priority value will apply to both legs of the trip.
 */

MomSend ( . . ., MOM_PRIORITY_NORMAL, . . .);
```

Notice that there is nothing very unusual about the above call to MomSend(). The priority argument to the function has been expressed as `MOM_PRIORITY_NORMAL`. By default, X•IPC MomSys will assign the specified priority to *both* legs of the message's trip.

The same would be true for any valid priority value. Just to make the point clear, consider the following example:

```
/*
 * Send a message having a slightly higher than normal priority for
 * its entire trip. This priority value will apply to both legs of the trip.
 */

MomSend ( . . ., MOM_PRIORITY_NORMAL + 1, . . .);
```

Here, too, the prescribed priority value `MOM_PRIORITY_NORMAL + 1` (i.e., 32,769) will be assigned to both legs of the trip. Priorities need not be expressed as offsets from the three pre-defined values (although it is often useful to do so). It is just as valid to express the above MomSend() call as follows:

```
/*
 * Send a message having a slightly higher than normal priority for
 * its entire trip. This priority value will apply to both legs of the trip.
 * (Same as previous example, except that integer value is used directly.)
 */

MomSend ( . . ., 32769, . . .);
```

In the next example, we will send a message that will have a *high* priority for getting through the system to the target app-queue, but will be assigned a *normal* priority relative to other messages once on the app-queue.

```
/*
 * Send a message having HIGH priority for the trip to the app-queue.
 * Message priority on the app-queue should be NORMAL.
 */

MomSend ( . . ., MOM_PRIORITY(   MOM_PRIORITY_HIGHEST,    /* Trip to app-queue */
                                 MOM_PRIORITY_NORMAL     /* Once on app-queue */
                              ),
          );
```

The ability to express two separate priority values for each leg of a message's movement is provided by the `MOM_PRIORITY()` macro. This macro is specified as the priority argument to MomSend(). The macro takes two valid priority values as its two arguments. The first value is the "trip to the app-queue priority" while the second value is the "trip through the app-queue priority".

Note, that the above example could have equally been coded as:

```
/*
 * Send a message having HIGH priority for the trip to the app-queue.
 * Message priority on the app-queue should be NORMAL.
 */

MomSend ( . . ., MOM_PRIORITY( 65536, 32768), . . .);
```

9.2.4.3 An Important Caveat

The above discussions regarding prioritization of messages *on* an app-queue all assume that the targeted app-queue was created having the attribute `MOM_ATTR_SET_PRIORITY` set when the app-queue was created. This attribute creates app-queues which have message prioritization as the app-queues' natural message sequence. If, however, the app-queue is created with the `MOM_ATTR_SET_TIME` attribute set (this is the default value), then such an app-queue has a FIFO natural sequencing of its messages and does not support any prioritization sequencing of its messages. Accordingly, any "on-queue" priority value expressed for a sent message does not advance the message ahead of previously inserted messages.

This caveat only affects the second phase (i.e., on app-queue) priority value. The first phase value (i.e., that which defines a message's urgency relative to other message moving through the system to the app-queue) continues to have its effect regardless of the natural sequencing of messages on the targeted app-queue.

9.2.5 CONCLUSION

X♦IPC MomSys provides a great deal of user control over message prioritization. Priorities may be assigned on a per-message basis and may range from 1 to 65,536. Furthermore, *X♦IPC* optionally allows the user to assign discrete priority values for the different phases of message's trip to its destination application. With these mechanisms it is possible to build distributed applications that are flexible and adaptive to the realities of traffic-flow contention so often found in real-world application messaging-based systems.

9.3 Appendix C: Message Specification in MomReceive()

9.3.1 WHAT IS AN APP-QUEUE?

Before addressing the topic of message specification from an app-queue, it is instructive to first understand *what is an app-queue*. An app-queue is a set of messages that are maintained according to certain logical sequences. This sequences is known as the app-queue's "natural" sequence.

9.3.1.1 "Natural" Sequence

Every app-queue that is created has, as one of its defining attributes, a natural sequencing of its messages. This is referred to the app-queue's natural message sequence. There are two possible natural sequences:

Time sequence

Priority sequence

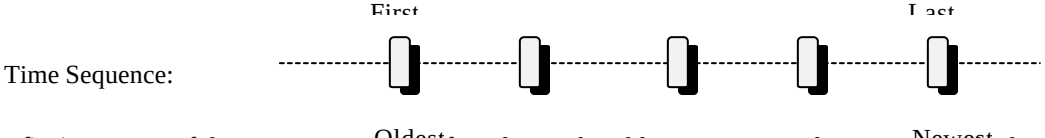
By default, an app-queue's natural sequence is the *time* sequence in which the messages arrive and are placed on the queue, i.e., the FIFO sequence. MomAttrSet() can be used to override thus default to create an app-queue whose messages are sequenced in *priority* sequence, i.e., highest priority at front of app-queue.

An app-queue's "natural" sequence defines the order by which messages are presented to users performing MomReceive() operations on that app-queue.

9.3.2 TERMINOLOGY

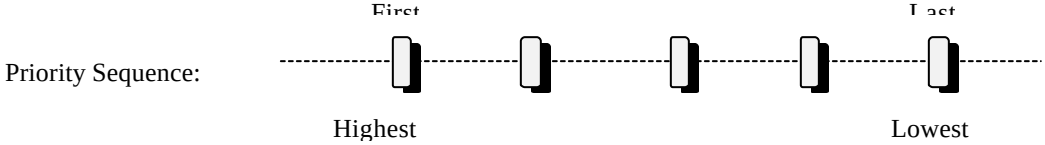
Correct message specification is dependent on a clear set of terms for defining the different messages on an app-queue.

The *time sequence* is the order that the app-queue's messages entered the app-queue, from oldest to newest.



The *front* (or *first*) message of the time sequence is referred to as the *oldest* message in the sequence. The *back* (or *last*) message in the sequence is the *newest* message. Each message arriving on an app-queue has a time stamp of when the message was enqueued. No two messages have the same time stamp.

The *priority sequence* orders messages from highest priority to lowest priority value.



The *front* (or *first*) message of the priority sequence is referred to as the *highest-priority* message in the sequence. The *back* (or *last*) message in the sequence is the *lowest-priority* message. Each message arriving on an app-queue has a priority assigned to it from the time it was sent. This priority governs the urgency of the message relative to other messages on the app-queue. Messages having the same priority are sequenced in FIFO order within that priority value.

9.3.3 POSSIBLE MsgSpecifier VALUES

The *MsgSpecifier* argument defined as part of the MomReceive() function plays an important role in the operation of that function. It tells MomReceive() which message is to be returned from the specified app-queue. The MomReceive() function description offers an extensive list of pre-defined values that may be used as the *MsgSpecifier* arguments to MomReceive(). These predefined values cover most typical message selection requirements. They are:

MOM_MESSAGE_FIRST	Retrieve the first message from natural sequence. If *Time,* the oldest message is returned. If *Priority,* the highest priority message is returned.
MOM_MESSAGE_LAST	Retrieve the last message from natural sequence. If *Time,* the newest message is returned. If *Priority,* the lowest priority message is returned.
MOM_MESSAGE_NEXT(*MsgId*)	Retrieve the next message from within natural sequence following the message identified by *MsgId.**. If *Time*, the next oldest message is returned. If *Priority,* the next highest priority message is returned
MOM_MESSAGE_PREV(*MsgId*)	Retrieve the previous message from within natural sequence following the message identified by *MsgId.**. If *Time*, the previous oldest message is returned. If *Priority,* the previous highest priority message is returned.
MOM_MESSAGE_REPLYTO(*MsgId*)	Retrieve the response message to the request message that was previously sent by MomSend() and identified as *MsgId*. *
MOM_MESSAGE_DIRECT(*MsgId*)	Retrieve the message identified by *MsgId*. *
MOM_MESSAGE_DIRECT_RMT (RmtNode, RmtInstance, RmtMsgId)	Retrieve a message based on its Remote identification: RmtNode is name of sender node RmtInstance is name of sender instance RmtMsgId is the MsgId assigned to the message when it was sent via the sender instance

(* Note: The message represented by MsgId must still be on the app-queue at the time of the MomReceive() call. This is typically accomplished by having performed an earlier call to MomReceive() in which the MOM_NOREMOVE flag was set. The MsgId returned from that call can serve as the "cursor" for subsequent MomReceive() calls.)

In fact, these predefined values hide a flexible message selection mechanism that can be used as a programming device by application programmers.

9.3.4 THE TWO COMPONENTS OF A "MsgSpecifier"

Selecting a message from an app-queue entails the specification of *two* pieces of information. They are:

The Sequence - The sequence defines what logical *sequence* of app-queue messages is to be used as part of the selection. The possible values are:

Natural - defined as **MOM_SEQUENCE_NATURAL**

Any - defined as **MOM_SEQUENCE_ANY** (any sequence will do; see DIRECT example below)

The Selected Message - This specifies which message, within the given sequence, is to be returned.

1. First Message - defined as **MOM_SELECT_FIRST**

The first message of the *Time* sequence is the oldest message.

The first message of the *Priority* sequence is the highest priority message.

2. Last Message - defined as **MOM_SELECT_LAST**

The last message of the *Time* sequence is the newest message.

The last message of the *Priority* sequence is the lowest priority message.

3. Next Message - defined as **MOM_SELECT_NEXT(*MsgId*)**

The next message after *MsgId*, moving from *first* to *last*, within the *Time* sequence

The next message after *MsgId*, moving from *first* to *last*, within the *Priority* sequence

4. Previous Message - defined as **MOM_SELECT_PREV(*MsgId*)**

The previous message before *MsgId*, moving from *last* to *first*, within the *Time* sequence

The previous message before *MsgId*, moving from *last* to *first*, within the *Priority* sequence

5. Direct Message - defined as **MOM_SELECT_DIRECT(*MsgId*)**

Returns the message identified by *MsgId* regardless as to what natural sequence that app-queue has

6. Direct Remote Message - defined as MOM_SELECT_DIRECT_RMT(*RmtNode*, *RmtInst*, Rmt*MsgId*)

Same as **MOM_SELECT_DIRECT**, but uses remote information about message

7. Response Message - defined as **MOM_SELECT_REPLYTO(*MsgId*)**

Returns a "response" message regarding a previously sent "request" message. Recall that the MomSend() and MomReceive() functions allow client and server programs to communicate with one another in an inquiry-response fashion, without the server having to know the identity of the client. (Refer to the "Client/Server Interaction" section of this guide for detailed examples of this.)

A client sends a request message to an app-queue being served by a server program. The server, after receiving the request message, sends a response message back. The client, in the mean time, issues a MomReceive() call specifying the `MOM_MESSAGE_REPLYTO(MsgId)` as the *MsgSpecifier* argument where *MsgId* identifies the originally sent request message.

9.3.5 PULLING IT TOGETHER

The *MsgSpecifier* argument to MomReceive() is actually the aggregate of two sub-arguments, namely the two ingredients just described:

Sequence

Message Selector

To understand how they come together, consider some of the *MsgSpecifiers* predefined by *X•IPC*.

MOM_MESSAGE_FIRST ::= {MOM_SEQUENCE_NATURAL, MOM_SELECT_FIRST}

MOM_MESSAGE_DIRECT(*m*):: = {MOM_SEQUENCE_ANY, MOM_SELECT_DIRECT(*m*)}

The remaining predefined *MsgSpecifier* values are similar in nature. You may examine them, as they are included in the `"xipc.h"` include files.

9.3.6 MSGSPECIFIER SYNTAX

It should be evident by now that the *MsgSpecifier* argument to MomReceive() is in fact a cover for two sub-arguments: {*Sequence, Message Selector*}.

Methods for defining one's own *MsgSpecifier* to MomReceive() are to define a new 2-value macro or simply to call MomReceive() with the two sub-arguments explicitly spelled out in the location of the *MsgSpecifier* argument. The syntax for the call to MomReceive() then becomes:

```
XINT
MomReceive(
        XINT *SourceAQid,
        XANY *MsgBuf,
        XINT MsgBufLen,

        /* The next two arguments define MsgSpecifier */

        XINT Sequence,          /* One of MOM_SEQUENCE_... */
        XINT Selector,          /* One of MOM_SELECT_... */

        ...
        ...
        )
```

9.4 Appendix D: MomStatus() and MomStatusWait() Function Definitions

The MomStatus() and MomStatusWait() functions are defined on top of other MomSys API functions. This appendix presents simplified forms of these definitions for demonstration purposes to provide a sense of how you can further extend the MomSys "verb-set" in a similar manner.

9.4.1 SAMPLE MomStatus() DEFINITION

MomStatus() calls MomInfoMessage() for getting data on a particular message-id. It then returns the status value within the *RetStatus* variable. The following is a simplified version of how MomStatus() is implemented:

```
XINT
MomStatus(MOM_MSGID MsgId, XINT *RetStatus)
{
    MOMINFOMESSAGE m;

    MomInfoMessage (MsgId, &m);
    *RetStatus = m.LatestStatus;
}
```

9.4.2 SAMPLE MomStatusWait() DEFINITION

MomStatusWait() calls MomEvent() for tracking a message up to a particular status. The following is a simplified version of how MomStatusWait() is implemented as a macro:

```
#define \
    MomStatusWait(MsgId, Status, BlockOpt)  \
        MomEvent(MOM_EV_MSG_STATUS((MsgId), (Status)), (BlockOpt))
```

10. INDEX