Computer Network Programming

Interprocess Communication Using IPC Sources

“Chat Via Message Queue”

Name: Mert DUMANLI

Student ID: 160315002 – NORMAL

0. Communication

Inter-process communication

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing. Methods for doing IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.

IPC is very important to the design process for microkernels and nanokernels. Microkernels reduce the number of functionalities provided by the kernel. Those functionalities are then obtained by communicating with servers via IPC, increasing drastically the number of IPC compared to a regular monolithic kernel.

Approaches

Method	*Short Description*	*Provided by (operating systems or other environments)*
File	*A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.*	*Most operating systems*
Communications file	*A unique, late-1960's form of inter-process communication, most closely resembling Plan 9's 9P protocol*	*Dartmouth Time-Sharing System*

Signal; also Asynchronous System Trap	*A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.*	*Most operating systems*
Socket	*Data sent over a network interface, either to a different process on the same computer or to another computer on the network. Stream-oriented (TCP; data written through a socket requires formatting to preserve message boundaries) or more rarely message-oriented (UDP, SCTP).*	*Most operating systems*
Unix domain socket	*Similar to an internet socket, but all communication occurs within the kernel. Domain sockets use the file system as their address space. Processes reference a domain socket as an inode, and multiple processes can communicate with one socket*	*All POSIX operating systems and Windows 10*
Message queue	*A data stream similar to a socket, but which usually preserves message boundaries. Typically implemented by the operating system, they allow multiple processes to read and write to the message queue without being directly connected to each other.*	*Most operating systems*

Anonymous pipe	*A unidirectional data channel utilizing standard input and output. Data written to the write-end of the pipe is buffered by the operating system until it is read from the read-end of the pipe. Two-way communication between processes can be achieved by using two pipes in opposite "directions".*	*All POSIX systems, Windows*
Named pipe	*A pipe that is treated like a file. Instead of using standard input and output as with an anonymous pipe, processes write to and read from a named pipe as if it were a regular file.*	*All POSIX systems, Windows, AmigaOS 2.0+*
Shared memory	*Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.*	*All POSIX systems, Windows*
Message passing	*Allows multiple programs to communicate using message queues and/or non-OS managed channels. Commonly used in concurrency models.*	*Used in RPC, RMI, and MPI paradigms, Java RMI, CORBA, DDS, MSMQ, MailSlots, QNX, others*
Memory-mapped file	*A file mapped to RAM and can be modified by changing memory addresses directly instead of outputting to a stream. This shares the same benefits as a standard file.*	*All POSIX systems, Windows*

Synchronization

An IPC mechanism is either synchronous or asynchronous. Synchronization primitives may be use to have synchronous behaviour with an asynchronous IPC mechanism.

Message queue

In computer science, message queues and mailboxes are software-engineering components used for inter-process communication (IPC), or for inter-thread communication within the same process. They use a queue for messaging – the passing of control or of content. Group communication systems provide similar kinds of functionality.

The message queue paradigm is a sibling of the publisher/subscriber pattern, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the publisher/subscriber and message queue models in their API, e.g. Java Message Service (JMS).

Overview

Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue. Many implementations of message queues function internally: within an operating system or within an application. Such queues exist for the purposes of that system only.

Other implementations allow the passing of messages between different computer systems, potentially connecting multiple applications and multiple operating systems. These message queueing systems typically provide enhanced resilience functionality to ensure that messages do not get "lost" in the event of a system failure. Examples of commercial implementations of this kind of message queueing software (also known as message-oriented middleware) include IBM MQ (formerly MQ Series) and Oracle Advanced Queuing (AQ). There is a Java standard called Java Message Service, which has several proprietary and free software implementations.

Implementations exist as proprietary software, provided as a service, open source software, or a hardware-based solution.

Proprietary options have the longest history, and include products from the inception of message queuing, such as IBM MQ, and those tied to specific operating systems, such as Microsoft Message Queuing.

There are also cloud-based message queuing service options, such as Amazon Simple Queue Service (SQS), StormMQ, IronMQ, and IBM MQ has a cloud-based managed queuing service.

There are a number of open source choices of messaging middleware systems, including Apache ActiveMQ, Apache Kafka, Apache Qpid, Apache RocketMQ, Beanstalkd, Enduro/X, HTTPSQS, JBoss Messaging, JORAM, RabbitMQ, Sun Open Message Queue, and Tarantool.

In addition to open source systems, hardware-based messaging middleware exists with vendors like Solace, Apigee and Tervela offering queuing through silicon or silicon/software datapaths. IBM also offers its MQ software on an appliance.

Most real-time operating systems (RTOSes), such as VxWorks and QNX, encourage the use of message queueing as the primary inter-process or inter-thread communication mechanism. The resulting tight integration between message passing and CPU scheduling is attributed as a main reason for the usability of RTOSes for real time applications. Early examples of commercial RTOSes that encouraged a message-queue basis to inter-thread communication also include VRTX and pSOS+, both of which date to the early 1980s.
The Erlang programming language uses processes to provide concurrency; these processes communicate asynchronously using message queuing.

Usage

In a typical message-queueing implementation, a system administrator installs and configures message-queueing software (a queue manager or broker), and defines a named message queue. Or they register with a message queuing service.

An application then registers a software routine that "listens" for messages placed onto the queue.

Second and subsequent applications may connect to the queue and transfer a message onto it.

The queue-manager software stores the messages until a receiving application connects and then calls the registered software routine. The receiving application then processes the message in an appropriate manner.

There are often numerous options as to the exact semantics of message passing, including:

- *Durability - messages may be kept in memory, written to disk, or even committed to a DBMS if the need for reliability indicates a more resource-intensive solution.*
- *Security policies - which applications should have access to these messages?*
- *Message purging policies - queues or messages may have a "time to live"*
- *Message filtering - some systems support filtering data so that a subscriber may only see messages matching some pre-specified criteria of interest*
- *Delivery policies - do we need to guarantee that a message is delivered at least once, or no more than once?*

- *Routing policies - in a system with many queue servers, what servers should receive a message or a queue's messages?*
- *Batching policies - should messages be delivered immediately? Or should the system wait a bit and try to deliver many messages at once?*
- *Queuing criteria - when should a message be considered "enqueued"? When one queue has it? Or when it has been forwarded to at least one remote queue? Or to all queues?*
- *Receipt notification - A publisher may need to know when some or all subscribers have received a message.*

These are all considerations that can have substantial effects on transaction semantics, system reliability, and system efficiency.

Standards and protocols

Historically, message queuing has used proprietary, closed protocols, restricting the ability for different operating systems or programming languages to interact in a heterogeneous set of environments.

An early attempt to make message queuing more ubiquitous was Sun Microsystems' JMS specification, which provided a Java-only abstraction of a client API. This allowed Java developers to switch between providers of message queuing in a fashion similar to that of developers using SQL databases. In practice, given the diversity of message queuing techniques and scenarios, this wasn't always as practical as it could be.

Three standards have emerged which are used in open source message queue implementations:

1. *Advanced Message Queuing Protocol (AMQP) – feature-rich message queue protocol, approved as ISO/IEC 19464 since April 2014*
2. *Streaming Text Oriented Messaging Protocol (STOMP) – simple, text-oriented message protocol*
3. *MQTT (formerly MQ Telemetry Transport) - lightweight message queue protocol especially for embedded devices*

These protocols are at different stages of standardization and adoption. The first two operate at the same level as HTTP, MQTT at the level of TCP/IP.

Some proprietary implementations also use HTTP to provide message queuing by some implementations, such as Amazon's SQS. This is because it is always possible to layer asynchronous behaviour (which is what is required for message queuing) over a synchronous protocol using request-response semantics. However, such implementations are constrained by the underlying protocol in this case and may not be able to offer the full fidelity or set of options required in message passing above.

Synchronous vs. asynchronous

Many of the more widely known communications protocols in use operate synchronously. The HTTP protocol – used in the World Wide Web and in web services – offers an obvious example where a user sends a request for a web page and then waits for a reply.

However, scenarios exist in which synchronous behaviour is not appropriate. For example, AJAX (Asynchronous JavaScript and XML) can be used to asynchronously send text, JSON or XML messages to update part of a web page with more relevant information. Google uses this approach for their Google Suggest, a search feature which sends the user's partially typed queries to Google's servers and returns a list of possible full queries the user might be interested in the process of typing. This list is asynchronously updated as the user types.

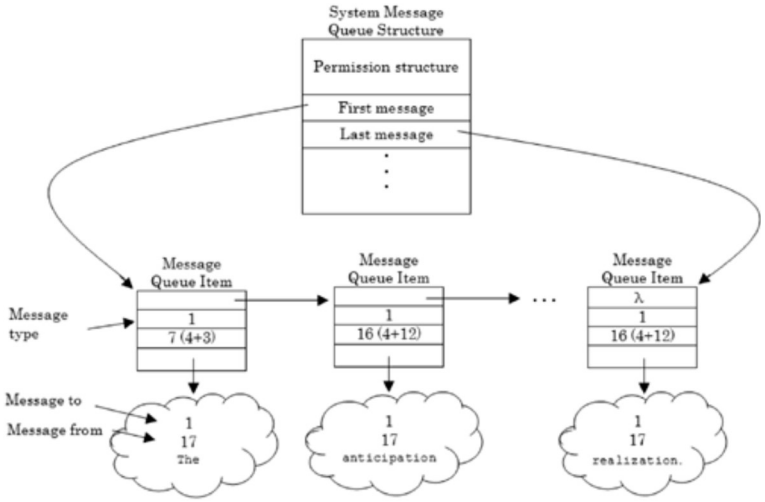
Other asynchronous examples exist in event notification systems and publish/subscribe systems.

- *An application may need to notify another that an event has occurred, but does not need to wait for a response.*
- *In publish/subscribe systems, an application "publishes" information for any number of clients to read.*

In both of the above examples it would not make sense for the sender of the information to have to wait if, for example, one of the recipients had crashed.

Applications need not be exclusively synchronous or asynchronous. An interactive application may need to respond to certain parts of a request immediately (such as telling a customer that a sales request has been accepted, and handling the promise to draw on inventory), but may queue other parts (such as completing calculation of billing, forwarding data to the central accounting system, and calling on all sorts of other services) to be done some time later.

In all these sorts of situations, having a subsystem which performs message-queuing (or alternatively, a broadcast messaging system) can help improve the behavior of the overall system.



1. *MessageQueueModel*

Implementation in UNIX

There are two common message queue implementations in UNIX. One is part of the SYS V API, the other one is part of POSIX.

SYS V

UNIX SYS V implements message passing by keeping an array of linked lists as message queues. Each message queue is identified by its index in the array, and has a unique descriptor. A given index can have multiple possible descriptors. UNIX gives standard functions to access the message passing feature.

`msgget()`

This system call takes a key as an argument and returns a descriptor of the queue with the matching key if it exists. If it does not exist, and the `IPC_CREAT` flag is set, it makes a new message queue with the given key and returns its descriptor.

`msgrcv()`

Used to receive a message from a given queue descriptor. The caller process must have read permissions for the queue. It is of two types.

- *Blocking receive puts the child to sleep if it cannot find a requested message type. It sleeps till another message is posted in the queue, and then wakes up to check again.*
- *Non-blocking receive returns immediately to the caller, mentioning that it failed.*

`msgctl()`

Used to change message queue parameters like the owner. Most importantly, it is used to delete the message queue by passing the `IPC_RMID` flag. A message queue can be deleted only by its creator, owner, or the superuser.

POSIX

The POSIX.1-2001 message queue API is the later of the two UNIX message queue APIs. It is distinct from the SYS V API, but provides similar function.

`mq_overview(7)`

This man page provides an overview of POSIX message queues.

C/C++ Code's reading/rewriting and writing in Linux

I wrote the C/C++ code in Bourne-again shell (bash) in linux operating system.

1. *I read codes and understood them. These codes are local.h, server.cxx and client.cxx.*
2. *Firstly, I changed local.h and I added two string for recipient and sender names. Also, I added required libraries.*
3. *The strings' name are name[20] and receivername[20].*
4. *After that, I have edited client.cxx and before I created "s_name" for I created it once for the senders. I did it with the help of the necessary methods. Therewithal, I created "r_name" for I created it once for the receivers. I wrote same operations in like s_name.*
5. *Finally, I added some libraries in server.cxx and I wrote:*

```
int pidID[5];
int q;
for(q=0;q<5;q++){
pidID[q] = 0;
}
int count = 0;
int id;
id = msg.msg_fm;
while(count < 5){
if(pidID[count]==0){
pidID[count] = id;
id = 0;
}
count = count + 1;
}
char pidnames[5][20];
int p;
for(p=0;p<5;p++){
pidnames[p] = 'A';
}
int count2 = 0;
char name[20];
name = msg.name;
int x;
while(count2 < 5){
x=strlen(pidnames[count2]);
if(x == 1){
pidnames[count2] = name;
name = 'A';
}
count2 = count2 + 1;
}
int k = 0;
int index;
while(k < 5){
if(strcmmp(msg.receivername,pidnames[k
]) == 0 )
index = k;
k = k +1 ;}
int l = 0;
int index2;
while(l < 5){
if(strcmmp(msg.name,pidnames[l]) == 0 )
index2 = l;
l = l +1 ;}
```

I kept both names and PIDs in memory in array of pidnames[], pidID[]. After, comparing the names, I found the recipient's name, followed by the PID of that person, and added a message queue.

Conclusion

As a result, we have seen the communication between processes using pid with client names. Each sent message is processed by the server and transmitted to the message queue. Since each client sees this queue that opened in kernel, messages can be forwarded easily. You can use ipcs -q command to get more information about message queues that are opened in Linux operating system.

Note

I created the r_name and s_name character variables in the "local.h" library. However, the first element assignment was not made. So when “client.cxx” is executed, it gives an error. In order to fix this, I assigned r_name and s_name as name values. But I had to use these names as variables.

References

- *https://docs.microsoft.com/tr-tr/windows/win32/ipc/interprocess-communications?redirectedfrom=MSDN*
- *https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvSW50ZXItcHJvY2Vzc19jb21tdW5pY2F0aW9u*
- *https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvTWVzc2FnZV9xdWV1ZQ*
- *https://linux.die.net/man/7/mq_overview*
- *http://www.civilized.com/files/msgfuncs.txt*
- *http://bilgisayarkavramlari.sadievrenseker.com/2008/12/01/strcpy-string-copy-dizgi-kopyalama*
- *https://stackoverflow.com/questions/4118732/c-array-assign-error-invalid-array-assignment*
- *https://www.yazilimgelistiricileri.com*