STEAM: AN ASYNCHRONOUS MESSAGING FRAMEWORK FOR ACTIVE OBJECTS

by

Paul Andrew Wierenga

B.Sc., University of Victoria, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. M. H. M. Cheng

ABSTRACT

This thesis describes a framework for building a distributed concurrent system based on the active object paradigm. An active object is an agent, such as a state machine, that only executes in response to messages it receives, and can only communicate with other agents by sending messages. Steam consists of a programming model, an application programming interface and a concurrent runtime system to support the model. Steam offers an efficient implementation of active objects. The implementation also supports location transparency, dynamic re-configurability, and a form of atomicity. Steam is designed as a generic framework suitable for application in a variety of domains. It can be used directly as a runtime system embedded within an application (eg. in a C program) or as a virtual machine for a concurrent programming language such as COOL.

Table of Contents

List of Tables

List of Figures

Listings

Acknowledgments

This thesis is dedicated to my wife, Shari, and children Robbie, Andrew, and Miranda. It has taken me a long time to complete this thesis, during which we have experienced many life changes. Thank you for putting up with me and for allowing me enough quiet time to write. Without your sacrifice, I could never have completed this work.

I am also indebted to many colleagues, for their time, input and ideas. Firstly, I would like to thank my supervisor, Dr. Mantis Cheng, for all his support and encouragement. Secondly, I would like to thank my examining committee: Dr. E. G. Manning, Dr. S. Ganti, and Dr. K. F. Li, for taking the time to read my thesis and offer constructive criticism. Next, I would like to thank Gordon O'Connell and Anthony Howe, for providing invaluable input into the design and implementation of Steam. I also must thank Bill Older for the discussions we had and his ideas for Crisp - it was truly an honor working under you. Lastly, I thank Voice Mobility Inc. for allowing me to use their lab equipment for conducting my experiments.

Financial Support for this research came from the Mathematics of Information Technology and Complex Systems (MITACS) of Canada, Master Solutions Inc, and Nortel.

As a final note, I must acknowledge the NHL Owners and Players' Association - Without the extended work stoppage and all the extra free time on Saturday nights, I am not sure I could have completed this thesis!

Chapter 1

Introduction

A software developer faces many challenges when implementing control software for a telecommunications device, such as a switch or router. Some of the challenges faced are due to its distributed nature, where agents executing on different processors must communicate and coordinate to achieve their goals. Other challenges have to do with concurrency: control software for such devices is typically highly concurrent, often having to satisfy real time requirements. Although actual data path functions such as switching connections and routing of data packets may be performed by hardware, the control path functions such as programming of hardware, execution of routing algorithms, and setup or teardown of connections is usually the responsibility of software. In addition to bandwidth capacity, such devices are often differentiated based upon the functions provided by control software and how fast such functions can be performed. Therefore, it is a reasonable expectation that such devices would be capable of high performance. Similarly, scalability and high availability are common goals for the software, when such devices are designed.

Steam is a re-usable framework for building distributed, concurrent systems, such as the control software for a switch or router. Steam is based on the idea of active objects communicating via asynchronous messages. For the purposes of this thesis, a distributed system can be defined as a collection of autonomous computing devices cooperating to achieve a common goal, with the restriction that such devices can only communicate with each other by sending messages. Such a system should support location transparency,

such that communications between logical components executing on different devices is indistinguishable from communications between logical components executing on the same device. This definition does not preclude a multiprocessor system as long as each processor runs autonomously and communicates via messages. In addition, Steam is a concurrent runtime system that can run continuously and is able to react to events in a timely manner. Furthermore, Steam is an architecture around which a distributed concurrent system can be developed. It is a general model that can be applied to numerous domains. In addition to the telecommunications domain, areas where Steam could be applied include the following:

- the messaging infrastructure to support a distributed supervisory control and data acquisition (SCADA) system;
- the engine for a distributed multi-user game [1];
- the modeling or simulation environment for developing and experimenting with distributed algorithms and protocols.

Steam was developed at the University of Victoria (UVic) in concert with COOL [2], a modeling language for developing applications based upon active objects and asynchronous messaging. COOL was developed at UVic by Gordon O'Connell. Steam evolved out of a concurrent runtime system called Crisp [3]. Crisp was developed at Nortel Networks Ltd. by Bill Older.

Applications based upon Steam are meant to run fast and efficiently, both in time and memory usage. Although the implementation of Steam described in this thesis is implemented on Linux with transport based upon User Datagram Protocol (UDP, RFC 758) over Internet Protocol (IP, RFC 791), Steam could easily be ported to other systems and transports.

To fully appreciate Steam and understand the rationale behind its design, it is important to provide motivation for why there is a need for such an approach. Employing a system such as Steam is but one approach that a software developer can take when developing such a system. Only from understanding some of the shortcomings of one of the more

traditional approaches, namely multithreading, can one appreciate the concepts promoted by Steam. To a large part, this motivation stems from the author's experience working on telecommunications software. Following this motivational discussion, the programming model for Steam will be presented. In addition to describing what design decisions have been made, the discussion will also address why certain design decisions were made.

1.1 Motivation

1.1.1 Objects and Threads

Using multiple threads of control (or tasks) within a common memory space (process) is one of the more conventional approaches to building a concurrent system. Using objects is a common method for architecting a program, providing both structure and abstraction. In the context of object-based programming, an object is an instance of a data type that encapsulates both state and behavior. State is represented by an object's member data, while behavior is represented by the object's member functions, often referred to as methods. In conventional object-oriented programming, objects are mostly passive. That is, the called method of an object executes in the same context (or call stack) as the calling object. Unfortunately, few implementations of objects are designed with concurrency in mind. To support concurrency, objects typically provide some form of synchronization within their methods, in order to permit multiple threads of control to safely execute their methods concurrently.

Developing applications based on threads and objects can be very expensive in terms of development effort, space efficiency, and time efficiency. In [4], Ousterhout provides a good summary for why multithreaded programming is complicated. There are a number of "classical" problems you can encounter with threads: race conditions, deadlocks, priority inversion, starvation, etc.... In addition, by adding objects to the mix, the software developer may also have to contend with additional complexities, such as the inheritance anomaly and reference counting, to ensure thread-safe construction and destruction of objects.

In addition to development complexity, threads can be expensive in terms of memory usage and program execution. Though threads may be considered light-weight when compared with (UNIX-style) processes, they still require kernel resources and memory to manage. With or without objects, each thread must have its own call stack which cannot be permitted to overflow during the course of the program's execution. As a general rule, it is safe to say that the larger the system and the more (circular) dependencies between components (objects, libraries, etc...), the larger a call stack could potentially grow, and the more memory must be allocated to its call stack. As well, the more threads you have in the system, the more such call stacks must be allocated. As an example, consider the default behavior under Redhat Linux on a 32 bit processor. In Linux, the default call stack size when you create a thread is 8 megabytes. If your maximum addressable memory for a process is 2 gigabytes, this leaves you with an upper limit of 256 threads. By reducing the default call stack size, you can create more threads, but care must be taken to ensure that call stacks are large enough not to overflow. Here, there is a trade off between how large and complex your system is and how many threads are required.

In addition to memory concerns, the time required to context switch and schedule the next thread to execute is often an issue. All the time spent during context switching and inside the kernel's scheduler represents time spent with no useful work taking place, at least from the application's perspective. Context switching can occur for a number of reasons. Depending of the type of scheduler, these context switches can be voluntary or involuntary (from the thread's perspective). To give the software developer a degree of control over the frequency by which threads execute, threads can often be assigned different priorities. Even with the power to assign priorities, it can still be a challenge to specify or even understand the runtime behavior of a complex multithreaded system.

1.1.2 Active Objects

As a response to the complexity brought about with the marriage between threads and objects, the concept of active objects was invented. An active object is a computational agent that communicates with other active objects by sending messages and is only activated

upon receipt of a message. Active objects differ from passive objects by the nature of how they are invoked. Active objects differ from passive objects, by de-coupling method execution from method invocation [5]. In other words, when a method of an active object is called, the execution of the body of the called method takes place in a different context than the calling object. In essence, each object has, or appears to have, its own thread of control. With active objects, there is a potential to decrease synchronization complexity [5], due to lack of shared resources. At the same time there is the potential to increase concurrency. Another significant advantage to decoupling invocation from execution is that execution does not all occur within a single call stack, and thus, the space allocated to a call stack can potentially be smaller.

Confining a unit of concurrency to an active object provides a useful way of partitioning a system. Instead of a system being composed of a large number of passive objects and threads intermixing, an active object based system is composed of a number of active objects which interact following basic rules dictated by message passing. Furthermore, by allowing objects to communicate via message passing semantics instead of function calls, many of the "classical" problems of multithreaded programs can be minimized or eliminated. For example, apart from the mechanics of how one active object sends a message to another active object, there should be no need for shared variables, and therefore, no need for semaphores, mutexes, etc.... This idea is similar to what can be achieved using message passing based systems, such as Send-Receive-Reply (SRR) [6], where the developer only needs to worry about the sequential nature of each agent, and need no longer be concerned about synchronizing on shared resources. As well, active objects represent a natural way to implement state machines, which are ubiquitous in control software for telecommunications devices, such as switches or routers.

When choosing to employ active objects as the fundamental building block for a system, one still needs to be concerned about their cost in terms of efficiency and scalability. A naïve implementation of active objects maps an active object to a thread in the underlying operating system. For a small number of objects, this approach seems to work quite well. If your application requires hundreds or thousands of active objects, the cost of allocating

so many threads may be too much. Alternatives are to use thread pools to map n active objects onto m threads, where m is typically much smaller than n.

1.1.3 Location Transparency and Distribution

Related to scalability of how active objects are implemented is scalability in terms of how easy it is to make a system distributed, such that it can take advantage of additional processing capability. Here, location transparency for message passing between actors becomes very important. If communications between active objects is semantically the same, regardless of their location, an application based on active objects can easily be distributed, and therefore, scales better than a system that can't be distributed as easily. Obviously, pointers and addresses to memory cannot be communicated in the contents of messages. If you need more performance out of an active object based system, where communications between active objects is location transparent, all you need to do is add more processors and migrate some of the active objects on to the new processor.

1.1.4 Message Passing

When evolving a concurrent system based on active objects into a distributed system, a choice has to be made between synchronous message passing and asynchronous message passing. Synchronous message passing systems (such as SRR), have merit because they typically come with ordering, guaranteed delivery, and built in flow control. If you use SRR between two threads in the same process, synchronizing the two threads for the duration of the communication is no more costly than using semaphores or mutexes. Here, when a thread sends, it remains blocked until it receives a reply from the receiver. When using SRR between threads in different processes, where processes may be physically distributed, synchronizing two threads can be much more costly. Here, the sending thread is not only blocked waiting for the receiver, but also waiting on network latency. It has been the author's experience that systems designed around SRR are typically thread heavy, where threads are often blocked waiting upon other threads. In a distributed control sys-

tem, blocking on remote agents could dramatically slow down and serialize an otherwise highly concurrent system. Therefore, for distributed active objects, asynchronous message passing is this author's preferred choice.

1.1.5 Ordering and Reliability

When implementing an asynchronous messaging system, a software developer has a choice between preserving order and providing reliability. Providing a system that is both reliable and order preserving would result in head of line blocking upon loss of a message. Head of line blocking can create back pressure on the party sending messages, which in turn would turn a (non-blocking) asynchronous messaging system into one that blocks.

For control applications, order of delivery of messages is arguably more important than providing end-to-end reliability [7]. For example, consider a simple SCADA system where three state machines coordinate in order maintain a consistent temperature for a room. When the temperature gets too high, a cooling system is enabled. When the temperature gets too cool, the cooling system is disabled. The first state machine, M, monitors the current temperature. The second state machine, E, is responsible for enabling or disabling the cooling system. The third state machine, C, is responsible for deciding whether the cooling system should be enabled as well as issuing an alarm if any errors occur. It is reasonable for the state machine M to periodically send status messages (reporting the current temperature) to state machine C. These messages need not be reliable; if one is lost, another will be sent shortly there after. If C does not receive a status message from M for a long time, C may choose to issue an alarm. If messages from M arrive out of order (eg: 3 hours later), state machine C could mistakenly decide to enable or disable the cooling system. When state machine C decides that the cooling system needs to be enabled, it sends a message to state machine E. Here, it is reasonable for C and E to adhere to a simple request/reply protocol, to ensure that not only did E receive the message but also that E succeeded at either enabling or disabling the cooling system. If E fails to respond (after several retries) or if E responds that it could not enable/disable the cooling system, C may choose to issue an alarm.

In the preceding example, the application (control) layer provides reliability, but only when it is needed. If the network itself is quite reliable and if the application layer ends up ensuring reliability anyway, it is reasonable for the messaging mechanism to not guarantee message delivery. One advantage of this approach, is that it simplifies the infrastructure that provides asynchronous messaging between active objects. Moreover, there should be less overhead, since copies of messages need not be maintained once a message has been transmitted.

Based on the preceding motivation, we are now ready to introduce the programming model for Steam.

1.2 Programming Model

1.2.1 Actors

The most basic abstraction in Steam is the active object, often called an actor. Like a state machine, actors in Steam embody state and behavior. The Steam runtime system maintains each actor's state across subsequent invocations. The set of messages that an actor is capable of receiving and sending represents its behavior. When a message is dispatched to an actor, the actor can alter its state, such that it can provide a different behavior the next time it is activated.

The computation an application performs is always done via processing of messages. Processing of messages is always performed in the context of an actor. In Steam, actors are the unit of concurrency. Scheduling of actors is based upon message load. The frequency by which messages are destined for a given actor define how frequently that actor is activated.

1.2.2 Timers

Built-in timers are used to drive time-driven invocations of actors; timers allow actors to react to the passage of time. The semantics of timers are similar to other actors, where

the timeout is communicated to the owning actor by sending it a timeout message. Two types of timers are provided: one-shot and periodic. When expired, a one-shot timer sends a timeout message and ceases to exist. A periodic timer, on the other hand, is like a one-shot timer that automatically re-arms itself upon expiration until it is explicitly cancelled. Periodic timers can be used to provide scheduled execution of actors.

1.2.3 Asynchrony

As alluded to above, methods of an actor are not invoked in the traditional sense. In Steam, invocation of methods is accomplished via asynchronous message passing. In fact, asynchronous message passing is the only mechanism of communication between actors. When one actor wishes to communicate with another actor (akin to invoking one of its methods), it does so by sending an asynchronous message. When an actor receives a message, the actor is activated to process the contents of the message[1]. To communicate results of a activation from the called actor back to the calling actor (if necessary), a second asynchronous message must be sent. There is no concept of a synchronous procedure call from one actor to another.

The programming model for Steam is highly asynchronous. The Steam runtime system provides an interface for one actor to create new actors, to create new timers, to cancel existing timers, and to send messages to actors. All such operations are asynchronous. What this means is the action of every call to the Steam runtime made by an actor is carried out in a different context from the calling actor. For example, the creation of an actor, creation of a timer, or cancellation of a timer is not actually carried out until the currently executing actor completes its invocation. Only in the context of the Steam runtime does the new actor or timer come into existence. Likewise, when an actor sends a message to another actor, the message does not actually get sent until the currently executing actor completes its invocation. Furthermore, the message will not be delivered until all the previously sent, but undelivered messages, are delivered first. If a message is sent to an actor

[1] The method that gets invoked on the destination actor could more appropriately be described as a message handler.

on a remote machine, the message must first be transmitted over the transport before it can be added to the set of undelivered messages on the remote machine.

1.2.4 Atomicity

Part of the reason why the programming model for Steam is asynchronous is so that it can support atomicity. When an actor in Steam is activated, it runs to completion without interruption. At the end of processing, an actor can choose to *commit*, *terminate*, or *abort*. Committing means that any state changes to the actor and any operations it may have performed will indeed be carried out. Terminating is the same as committing, except that the actor will cease to exist from that point forward. Aborting has the semantics of undoing any state changes and operations performed during an invocation, returning the object to the state it was in prior to receiving the message. Because operations are not carried out by the Steam runtime until the invocation completes, Steam can provide a form of atomicity, also referred to as *local checkpointing* [8].

Atomic semantics were chosen by Steam in order to provide error handling capabilities, similar to exception handling in more conventional object-oriented languages. When an invocation is aborted, the message currently being processed is discarded and treated as though it was "lost". Aborting allowing actors to recover gracefully in situations that might otherwise become quite complicated. For example, if a problem was detected just after an actor sends a message (but before it finished its activation), without atomicity, it would be very difficult for an actor to undo or cancel the resulting behavior when the destination actor receives and reacts to the messsage.

1.2.5 Distribution and Messaging

Steam supports seamless distribution; method invocation (if we can still call it that) between actors in the same memory space is the same as method invocation between actors in different memory spaces. Location transparency is achieved by a simple object identification scheme where each message that is sent from one actor to another is labeled with a

source and destination identifiers that are globally unique.

For two actors to communicate in a distributed system, they need only learn the identifier of their counterpart. Actor identifiers can be freely communicated in the contents of messages, thus allowing for dynamic re-configurability. For example, actor α may know actors β and γ. Actor β only knows α. Actor α can send β a message containing the identifier for γ. Then, actor β can start to communicate with γ.

In addition to source and destination identifiers, each message that is sent by one actor to another contains a unique method identifier and optional arguments. The unique method identifier is used by the activated actor to distinguish the different messages it might receive. The method combined with optional parameters can be considered a variant record for passing data within the message. It is the responsibility of the source actor to marshal typed data into a message and it is the responsibility of the destination actor to un-marshal the message back into typed data. Type safety of messages is beyond the scope of Steam, though provides motivation for using languages such as COOL.

1.2.6 Ordering and Reliability

Messages sent between actors are order preserving, but unreliable. For consistency between messages sent to local and remote actors, it is assumed that the Steam transport is order preserving. Message delivery adheres to a weak form of order preservation. That is, if actor α sends two messages to actor β, the first message is guaranteed to arrive before the second message, assuming neither message is lost. Ensuring order of delivery makes the programming model for Steam simpler. For example, two communicating actors need not worry about having to handle messages that occur out of order, unless of course messages are lost. This form of order preservation is referred to as weak since it says little about the order of messages sent to two different actors. For example, if an actor α sends two messages, one to actor β and one to actor γ, there is no guarantee that β will be activated before γ. If β and γ are located on the same instance of the Steam runtime, however, Steam will provide such a guarantee. If β and γ are located on different devices/machines, the order of delivery of messages is at the mercy of the underlying transport.

Delivery of Steam messages is not guaranteed. Steam makes the assumption that the underlying communications medium (memory, bus, or network) is highly reliable, and that the loss of a message is a very rare occurrence. Handling lost messages, if required, is the responsibility of the active object sending the message. For many applications, such as information dissemination (eg: process variables) in a data acquisition system, the loss of a few messages may be tolerated, especially if similar information is retransmitted on a regular basis. In such a system, loss of a few messages is more easily tolerated than out of order delivery of messages. For a control application where lost messages cannot be tolerated, a simple request-reply protocol can be implemented where retransmission driven by timers achieve required reliability. Relying upon the application layer to acknowledge control messages is not an unreasonable expectation. Furthermore, since an aborted actor activation results in a message being "lost", providing reliable delivery of messages does not readily make sense in the case of Steam.

1.2.7 Single Thread of Control

Steam runs as a single thread of control, where each actor runs to completion. All active objects running on the same node share the same thread of control. This simplification allows actors to be considered "feather-weight" when compared with threads. Since all actors share one thread, there need only be one call stack. Since each actor runs to completion, the only context information that needs to be maintained between actor activations is the actor's state itself. The call stack for an actor is essentially empty at the completion of an activation. There is no need for Steam to maintain separate call stack or program counter for each actor. By mapping all active objects to a thread pool of size one, there is no need for the kind of context switching and scheduling present in traditional multithreaded systems.

Since Steam runs as a single thread of control, if an actor misbehaves or consumes too much processing time, the invocation of other active objects will be impacted. To address this, Steam assumes that processor bound computations can be either be split up over multiple actor invocations or avoided altogether. Furthermore, Steam assumes that block-

ing activities within an invocation can also be avoided. That is, during an actor invocation, the actor does not perform any activities that might block the main thread of the process hosting the application (if running on a host operating system such as Linux).

1.2.8 Scalability

A hallmark of Steam is its ability to scale, and at the same time efficiently dispatch messages, route messages, and provide timers. Each instance of the Steam runtime is called a node. Steam is designed to support a large number of networked nodes, each capable of supporting a large number of actors and timers. For example, in the default 32 bit configuration for actor object identifiers, Steam supports up to 4096 actor instances, up to 4096 timers, up to 256 different types of actors, and up to 256 nodes. By increasing the size of object identifiers or adjusting how their 32 bits are used, the number of actors, timers, and nodes can be scaled up or down, as required.

As alluded to above, actors are extremely light-weight compared to threads, demanding less memory and resulting is less overhead. In addition, with location transparency, Steam applications developed for a single node can easily be reconfigured to run on multiple nodes, taking advantage of any additional processing resources.

1.3 Related Work

Numerous languages and systems based upon active objects have been developed. References [9] , [10], and [11] contain summaries of many of these languages and systems. What follows here is a brief survey of select systems, focusing primarily on systems that most closely relate to Steam. We start by introducing actor theory developed by Hewitt and Agha. This is followed by a discussion of systems designed to support active objects. In particular, we focus on systems that can be classified as runtime systems, have implementations in C or C++, and are suitable for Steam's application domain. The systems discussed include Act++ [12], Broadway [13], Quantum Programming [14], and Crisp [3].

1.3.1 Actor Theory

The active object concept originated with Hewitt's actor theory, where Hewitt believed artificial intelligence could effectively be modeled by "a society of communicating" actors [15]. Hewitt defined actors as "computational agents which carry out their actions in response to incoming communications" [16]. The actions that actors can perform are all asynchronous. These actions include sending communications to other actors, spawning new actors, and specifying a replacement behavior. The language and runtime system originally designed to support the actor concept was called PLASMA [15]. PLASMA was based upon Lisp. Specifying a replacement behavior via the *become* primitive is the means by which an actor can change its behavior (or state). Once an actor has processed a message and optionally specified its replacement behavior, it ceases to exist. If a replacement behavior was specified, it becomes the recipient for the next message to arrive for that actor.

Agha expanded the Actor model, focusing more on the domain of concurrency and distributed systems. Agha's intention was to apply the Actor model to the realm of programming a massively parallel distributed system [17]. In such a system, the replacement behavior for actors could occur in parallel to the original behavior of the actor. For example, if two messages arrive for an actor, the current (original) actor receives the first message. Once the original actor defines a replacement behavior, the replacement behavior can potentially process the second message, concurrent to the original actor. In theory, the Actor concept supports a pipelining ability of having both the original actor and its replacement behaviors' computations executing concurrently. On a system made up of a large number of processors, this concept provides a means for achieving a high degree of true parallel processing.

Another concept introduced by Hewitt and Agha is that of an insensitive actor. An insensitive actor is one that must delay before it can compute its replacement behavior. This delay is required because it may need to communicate with other actors before it can define its replacement behavior. While it is delayed, all messages to this actor are buffered until it receives the particular communication it is waiting for so that it can define its replacement behavior. While it is in this delayed state, it is called insensitive, as it is not particularly

responsive to all communications it may receive from other actors.

Steam has a number of similarities with the Hewitt/Agha Actor model. Firstly, both models support asynchronous buffered messages as the only means for actors to communicate. In the Actor model, each actor has a unique mailbox, which is the handle by which one actor can address another actor. Likewise, in Steam, each actor has a globally unique identifier that is used for addressing messages. Both the Actor model and Steam support dynamic re-configurability, where by the identities of other actors can be communicated to third parties in the contents of messages. As well, both models seamlessly support distribution and the idea of location transparency. Sending messages to local actors is semantically equivalent to sending messages to remote actors.

Steam also has a number of differences from the Hewitt/Agha Actor model. One of the most striking differences is the degree parallelism each model is intended to support. The Actor model was designed with a high-degree of concurrency in mind. Replacement behaviors for actors can potentially run in parallel with the original behavior. Moreover, multiple actors can run concurrently. To Agha, the Actor concept is a means by which hardware architectures with parallel processing capabilities can be exploited. Steam, on the other hand, is single threaded. When each Steam actor is activated, it runs to completion. The rapidity by which actors can be activated (ie: messages can be dispatched) is how single-node Steam approximates concurrency. Furthermore, at any one time on a given Steam node, there can be only one actor currently active. Steam is not intended to take advantage of multiprocessor systems. Instead, Steam is targeted at embedded systems and more conventional single-processor machines that exist within a networked environment.

Another important difference between Steam and the Actor model relates to the how messages are treated in both models. Although both models support asynchronous messages, the Actor model does not make any guarantees about the order of deliver. However, the Actor model does guarantee that all messages sent will eventually be delivered, though messages are subject to arbitrary delays. Steam does not guarantee message delivery, but does guarantee the order of delivery for messages sent between actors. Furthermore, Steam does not support any mechanisms for enforcing constraints on messages processed, nor

does Steam provide support for insensitive actors. Steam tries to keep things simple, where messages are always dispatched in the order they are sent, regardless of whether the recipient is ready to receive them or not.

1.3.2 ACT++

Kafura et al have built upon the foundations laid by Hewitt and Agha by developing ACT++, a C++ framework for programming with Actors [12]. The focus of ACT++ is a class library for concurrent programming. Unlike Steam, it is does not support actor communications in a distributed environment.

Actors in ACT++ are objects created by instantiating its actor class, passing in a reference to its (initial) behavior object. Message processing in ACT++ is accomplished by spawning one thread per message. Furthermore, Actors can be created with more than one thread of control to "enable the possibility of concurrent message processing by allowing two or more behaviors of the actor to process request messages simultaneously." To change its current behavior, ACT++ provides a *become* operation for defining a new (replacement) behavior object.

An important difference between Steam and ACT++ is how actors relate to message queues and message processing. In ACT++, actors have the ability to define new message queues and place constraints on message arrival order. Steam provides no such capabilities; in Steam message ordering between any two actors is always preserved. ACT++ provides two mechanisms for constraining message processing: behavior sets and CBoxes.

In ACT++, message processing by an actor is subject to its current behavior set. Only messages destined for behaviors that are currently present in an actor's behavior set can be dispatched. By permitting an actor to define its behavior set based upon its current state, the actor has the power to selectively postpone certain messages from being delivered until it decides that it is ready for them. Behavior sets can be used as a means for specifying synchronization constraints that can be safely over-ridden in an inheritance situation, thus addressing the inheritance anomaly [9].

The concept of CBoxes is another means of placing constraints of message arrival

order in ACT++. A CBox is type-safe message queues. Actors can dynamically create CBoxes. After sending a message to another actor, the current actor can block waiting to receive a message on a particular CBox before proceeding with processing of other messages, thus constraining the order of message processing. CBoxes permit an actor to distinguish between the handling of request messages and handling of reply messages. They are means by which insensitive actors can be implemented. In addition to contrasting with Steam, this approach also contrasts with the Hewitt/Agha model as implemented in PLASMA, where all messages for an actor (requests and replies) share the same queue.

1.3.3 Broadway

Broadway was developed by Sturman [13] to support his research into fault tolerance and the design of systems that adapt to faults. Broadway is an actor based runtime system developed in C++. Actors in Broadway inherit from Broadway's actor class. The actor class provides a mail queue for each actor, handles transmission of messages and supports actor scheduling. Like Steam, the Broadway's scheduling strategy is based on a "single thread of execution shared all actors on a node," referred to as a Daemon. Unlike Steam, Broadway applies fairness to its actor scheduling. Each actor on a node "processes one method invocation (message) in a round-robin manner". Between actor invocations, the Daemon also handles intranode communications. Methods in this system "are atomic actions and may not be blocked after execution commences". In addition to providing a *become* mechanism for specifying replacement behaviors, Broadway also provides an actor migration ability for actors to migrate from one node to another. Steam neither supports replacement behaviors nor actor migration. In Steam, changing an actor's behavior is accomplished by changing its local state.

Broadway was later used by Frolund as the execution environment to support his research into coordinating distributed (active) objects [10]. Frolund offers an alternative to behavior sets for dealing with synchronization constraints. In [10], language constructs called "synchronization constraints" and "synchronizers" are combined with the Actor model to enforce a desired ordering upon messages received by an actor.

1.3.4 Quantum Programming

Quantum Programming (QP) is a programming methodology for embedded systems, where a design specified via Statecharts can be implemented in terms of a framework based upon active objects [14]. Implementations of QP framework are offered in both C and C++. QP, like Steam, supports the idea of a single thread of control, which is shared by all active objects. Alternatively, the runtime system for QP can be configured such that each active object has its own thread. Unlike Steam, QP uses event passing rather than message passing. Events do not contain the identities of either the sender or receiver. An active object is invoked when it receives an event. Like Steam, active objects in QP obey run to completion semantics when processing an event. Instead of identifying the recipient of an event, QP promotes a loose coupling of active objects, where events are communicated between active objects via a publish and subscribe model. Active objects register for events they are interested in. As well, active objects can publish events that other active objects may be interested in. QP is targeted at an embedded system and does not directly support event passing between active objects in a distributed environment.

1.3.5 Crisp

The system most closely related to Steam is Crisp, a proprietary system developed at Nortel Networks Ltd [3]. Crisp is a reusable framework upon which control software could be developed. The target application for Crisp is a telecommunications device such as a network switch or router. Like Steam, Crisp is a single threaded runtime system to support active objects communicating via asynchronous messages. Speed and simplicity were the objectives behind Crisp. Although Crisp supports the notion of atomicity for operations such as sending messages, it lacks the ability to restore an active object's state when an invocation aborts. Crisp's implementation language is based on C++, where active objects are mapped into C++ objects. Consequently, providing the ability to restore an active object's state posed challenges that prevented Crisp from supporting atomicity to the same level as Steam. Furthermore, Crisp relies upon an interface definition language (IDL) and compiler

to generate much of the message handling code. Type safety was provided by a combination of IDL and the C++ compiler. Eventually, Crisp evolved to include multithreading to address the problem of actors taking too long to process messages. In multithreaded Crisp, each actor invocation can take place in a separate thread allocated from a thread pool, such that concurrent invocations can occur so that one computationally intensive invocation does not impact other actors.

1.4 Contributions of This Research

As discussed in the previous section, many active object based systems have been proposed and implemented. Steam can be viewed as an enhancement to the Actor model; Steam implements a *Timed Actor model* [8], where messages can be triggered by the passage of time as well being sent by regular actors. Steam offers an implementation of actors that is targetted for the domain of distributed and concurrent systems, where asynchrony, speed and efficiency are of the utmost importance. Steam's model of actors supports dynamic re-configurability, is scalable, and achieves location transparency. Furthermore, Steam supports a notion of atomicity that is more complete than that offered by Crisp.

Steam is a reference implementation for the COOL virtual machine. As such, this thesis contains a discussion of the design and implementation decisions that shaped both Steam and COOL. This thesis offers the reader insight into how an efficient runtime system to support a concurrent programming language can be constructed.

1.5 Thesis Outline

The organization for the remainder of this thesis is as follows. Chapter 2 addresses some of the more practical aspects of Steam. Topics covered by this chapter include the architecture of a Steam application, background on COOL, a description of the interface to the Steam runtime system, an overview of how booting and discovery are accomplished in Steam, and a discussion of some issues faced by the Steam application programmer. Chapter 3 focuses

on the internal architecture of Steam: actor representation, timer representation, transport, asynchronous messaging, and message dispatching mechanism. Where relevant, design issues that lead to a certain implementation are also discussed. Chapter 4 provides a few case studies that demonstrate Steam in action. A summary of its performance characteristics is also provided. Chapter 5 contains concluding remarks and directions for future research.

Chapter 2

Basic Concepts

This chapter is divided up into six parts. Section 2.1 outlines the architecture of Steam application. Section 2.2 provides a background on COOL. Section 2.3 describes the interface to Steam, providing an overview of the Steam application programming interface (API). The purpose of the overview is to outline how Steam's key concepts are represented by functions and data structures. References to COOL are included to provide some insight into how concepts from COOL are mapped into Steam. Sections 2.4 and 2.5 discuss the process of booting and discovery, outlining both the single node and multi-node cases. Section 2.6 discusses issues faced by the Steam application designer and provides insight into why one might opt to use COOL rather than "raw" Steam.

2.1 Architecture Of a Steam Application

A non-distributed Steam application consists of a single Steam node, which provides the execution environment for all actors and where all communication is between actors existing on the one node (see Figure 2.1).

On the other hand, a distributed Steam application consists of a number of Steam nodes, each providing an execution environment for actors, but where communication can either be between actors existing on the same node or between actors existing on different nodes (see Figure 2.2).

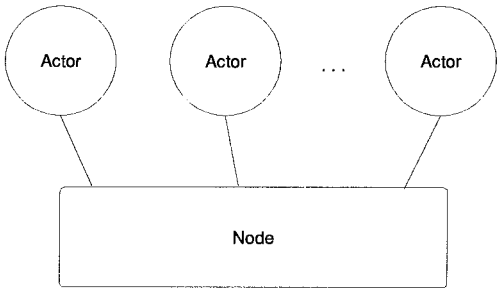

Figure 2.1: Single Node Steam Application

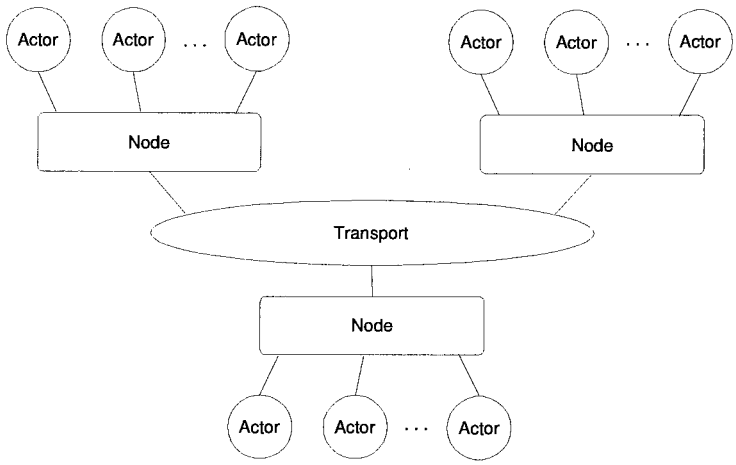

Figure 2.2: Multi Node Steam Application

From an actor's perspective, message distribution is transparent; the logic of an actor should not care where its peer actors are executing. The runtime environment for Steam provides location transparency, where communication between actors executing on the same node is indistinguishable from communication between actors executing on different nodes.

When a Steam application is compiled, a single executable is produced. Steam supports the ability for different nodes to start up with different actors. Within a Steam executable, there are three categories of actors: Engine, Mom, and application-level actors (see Figure 2.3). Engine and Mom are special actors utilized by the booting process of Steam. Application actors, on the other hand, are the actors that define the application's

main computation.

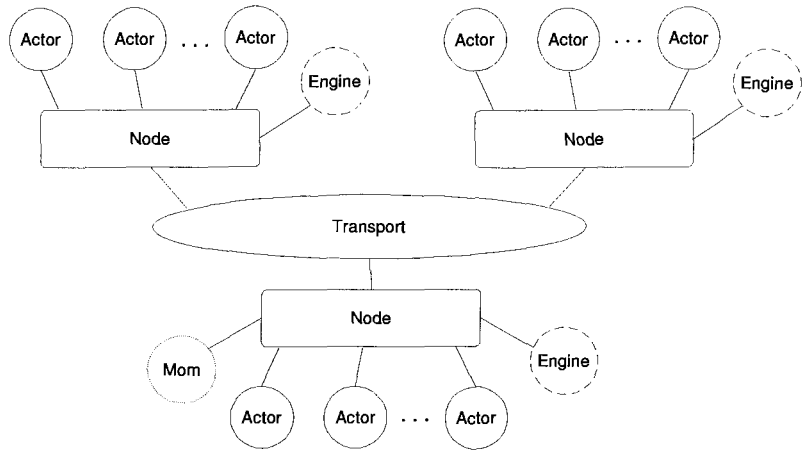

Figure 2.3: Multi Node Steam Application With Engines

The Engine embodies an actor representation for the Steam runtime system itself. There is only one Engine actor per node. The function of the Engine actor is to boot the node by creating the initial application actors for that node.

Mom represents a special singleton actor; there is only ever one instance of Mom across all nodes, regardless of whether the application is distributed or not[2]. Mom has a well known actor identifier. The node hosting Mom is responsible for assigning node identifiers to each node. Only after an Engine has been assigned a node identifier is it able to create its initial set of application actors.

An important aspect of Steam is how associations between actors are formed and how such associations are dynamically reconfigurable. When an actor is created, it only knows about Mom, itself, its creator, and any actors that it may have created. An actor's creator, the actor itself, and the actors that it creates can be considered its direct ancestor tree (see Figure 2.4). Once an actor starts to receive messages, it may learn identities of other actors.

The Engine is the root of all ancestor trees, since it is responsible for creating the initial set of application actors on a given node. Through actor identifiers passed within

[2]It is recognized that providing a single Mom represents a single point of failure. When Steam was designed, fault tolerance was not its primary goal.

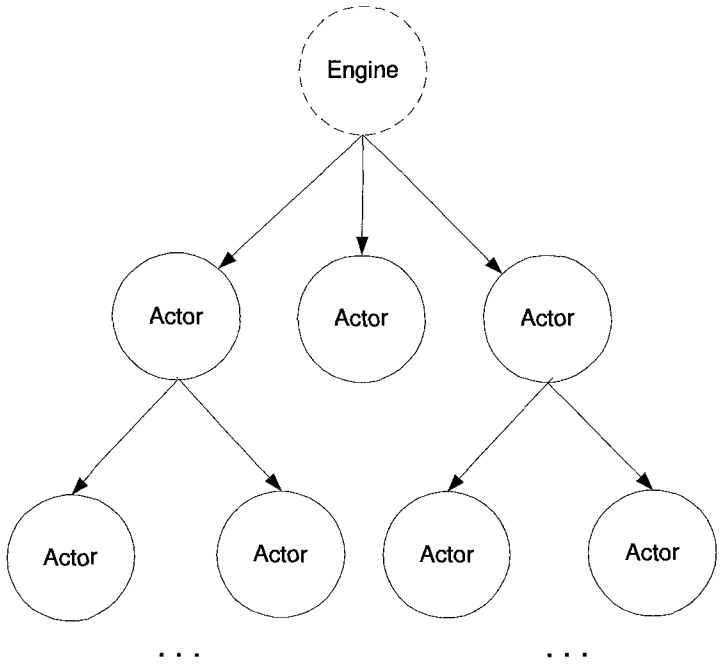

Figure 2.4: Actor Ancestor Tree

messages sent between different actors in an ancestor tree, actors can dynamically develop new associations. To aid with discovery of services offered by remote actors, it is a common practice for Mom to provide a basic name server function, so that actors with no common ancestor can register and lookup actor identifiers, without regard to locality. With such an extension (which is beyond the scope of Steam itself), combined with the fact that Mom is responsible for assigning node identifiers to each Engine, Mom can be considered an indirect common ancestor for all actors in the system.

2.2 A COOL Primer

Steam was developed in concert with COOL. COOL is a concurrent programming language for expressing communication and coordination between active objects. COOL was initially conceived as a tool to make programming Steam easier and safer, from a type safety perspective. COOL provides a natural way to program active objects by offering

language support for many of the concepts in Steam, including actors, asynchronous messaging, timers, and, atomicity. The language support provided by COOL includes syntax checking, semantics checking, and code generation. COOL itself can also be viewed as a language for realizing actor behavior formally described in Asynchronous Actor Algebra (ACUBE)[8].

When using COOL, the programmer writes the source code in the COOL language. COOL is not meant to be a general purpose programming language. Any functionality that cannot be expressed in terms of COOL is accessed via external C/C++ functions, which can be called from the COOL source. The product of compiling COOL source code is C/C++ code emitted by the COOL compiler. The emitted C/C++ code is then compiled with a C/C++ compiler and subsequently linked with Steam (as well as any other libraries or object files) to produce an executable application. The C/C++ code emitted by the COOL compiler is similar to what a programmer might write if he were to develop an application using "raw" Steam. COOL looks after mapping all COOL level constructs and syntax into C/C++ code, which depends upon services provided by Steam.

The primary goal of COOL is to provide a "high-level description language for specifying the interfaces, behaviour and coordination" of actors [2]. In addition to providing data types to represent actors, COOL also provides a data type for timers as well as a number of more primitive data types. What follows is a brief introduction to the COOL language. Here we focus on key concepts that are supported by the Steam runtime system. Many advanced features of COOL are omitted. In later chapters, COOL source code listing will be presented to show how experimental results were attained. [18] and [19] provide a more complete introduction to the syntax and semantics for COOL. [20] provides a description of how COOL source is translated into the C/C++.

2.2.1 Defining Actors

Actors are defined in terms of an actor class, which serves as a template from which actors can be instantiated. As part of an actor class definition, the programmer specifies state

variables, a messaging interface, and actor class attributes[3]. To illustrate each of these aspects, we present an actor that represents a *stack* (see Listing 2.1). This implementation of a stack is not particularly useful, as it can only receive messages. Later, we will enhance it so that the element at the top of the stack can be retrieved via message passing.

In this example, *buffer*, *index*, and *sizeOfStack* represent the actor's state variables, and *push*, *pop*, and *init* represent its messaging interface (ie: the messages it can receive). An enumerated type is also defined, which lets us specify the dimension of the buffer array as a constant. When the actor class *Stack* is first created, it must be sent an *init* message which specifies how high the stack is allowed to grow. When this actor is activated with a *push* message, the element passed as part of the message parameters is inserted at the top of the stack. Likewise, when this actor is activated with a *pop* message, this actor responds by removing the top element from the stack.

The message handling code for each message in COOL concludes with either *commit*, *terminate*, or *abort*[4]. If none of these keywords is specified, *commit* is assumed. These keywords have the same semantics as defined previously by Steam. In our example, all error handling is handled via *abort*. If index is ever out side of the range *0..sizeOfStack*, the message invocation aborts. Otherwise, message invocations always commit.

In addition to defining local state (variables) and a messaging interface, COOL allows the programmer to define actor class attributes. Attributes are a way of assigning certain properties to an actor. Possible attributes include *autoboot*, *bootable*, *mom*, *engine*, *immortal*, *atomic*, and *instances*. Attributes are described in Table 2.1. In our example, Stack is defined to be *atomic*.

[3] State labels and actor methods can also be specified. We omit discussion of these as they are COOL-only concepts and are not important to Steam

[4] A message handler in COOL can also conclude with *becomes*. The *becomes* keyword is unique to COOL; it must be used in conjunction with a state label. State labels are a way to group messages that an actor might receive into conceptual states. State labels exist at the level of COOL only and are transparent to Steam. State labels allow the actor to restrict the messages it can handle. Transitioning from one conceptual state to another is via *becomes*, which is like *commit* except that a new conceptual state is specified.

Table 2.1: Actor Class Attributes

Attribute	Description
bootable	A *bootable* actor is one that can be configured to be the main actor that gets instantiated by an Engine when a node initializes. In other words, a bootable actor is one that is able to boot an application.
autoboot	A *autoboot* actor is one that is instantiated automatically by each Engine as each node initializes. Using *autoboot* is good way to ensure certain services start on every node. More than one actor can be tagged *autoboot*.
mom	An actor tagged as *mom* results in that actor assuming the role of Mom.
engine	An actor tagged as *engine* results in that actor assuming the role of Engine.
immortal	An *immortal* actor is one that cannot terminate; *immortal* disables the *terminate* operation. compiler.
atomic	An *atomic* actor is one that has the content of its local state preserved if an invocation aborts. Without the *atomic* attribute, the *abort* operation is disabled.
instances	The *instances*attribute is used for specifying the maximum number of instances for a particular actor class.

Listing 2.1: COOL Source for the Stack Actor

```
enum Defines { MaxSizeOfStack = 100};
actor class Stack [atomic] {
  int buffer[MaxSizeOfStack];
  int index = 0;
  int sizeOfStack = 0;

  message init(int size) {
    if (size != 0 && size < MaxSizeOfStack) {
      sizeOfStack = size;
      index = 0;
      commit;
    } else {
       abort;
    }
  }

  message push(int el) {
    buffer[index] = el;
    index = index + 1;
    if (index < sizeOfStack) {
      commit;
    } else {
      abort;
    }
  }

  message pop() {
    index = index - 1;
    if (index < 0) {
      abort
    } else {
      commit;
    }
  }
}
```

Listing 2.2: COOL Source for Instantiating a Stack Actor

```
Actor A
{
  ...
  Stack s = none;
  message () start {
    s = new Stack;
  }
}
```

2.2.2 Actor Operations and Actor References

Actors are created via the *new* keyword. Creating an actor allocates its resources and initializes its local state. For the newly created actor to be activated, it must be sent a message. Often we refer to an actor that has been created as an instantiation of a particular actor class. To illustrate this, consider the example in Listing 2.2, which shows one actor creating an instance of *Stack* presented earlier. The local state variable *s* of type *Stack* is a reference to the instantiation of the *Stack* actor class.

Each created actor has a globally unique identity or reference. Actor references are used as the destinations when sending messages between actors. In addition to sending of messages, operations allowed on actor references include tests for equality and assignment. Furthermore, actor references can freely be passed around in the contents of messages, allowing actors to form new associations.

COOL has four pre-defined keyword that represent special actor references: *self*, *owner*, *mom*, and *none*. *self* is an alias for the current actor. *owner* is an alias for the actor that created the current actor. *mom* is a reference to the actor representing Mom, as described earlier. *none* is a universally invalid actor that is the same regardless of the node. *none* is a constant that refers to a non-existent actor; sending messages to *none* is a null operation.

Actors can only be destroyed by ending an invocation with the *terminate* keyword. Listing 2.3 shows how actor destruction is accomplished by our Stack example.

Listing 2.3: COOL Source for Actor Termination

```
actor class Stack [atomic] {
  ...
  message stop(){
    terminate;
  }
}
```

Listing 2.4: COOL Source for Sending Messages

```
Actor A {
  Stack s = none;

  message start(int el) {
    s = new Stack;
    init(10) => s;
  }

  message replaceTopOfStack(int el) {
    pop() => s;
    push(el) => s;
  }
}
```

2.2.3 Sending Messages and the Message Data Type

In the above example, Stack can only receive messages and modify its local state; it cannot send any messages. For an actor to interact with other actors, it must be able to both send and receive messages. The syntax for sending messages is via the '⇒' operator. The message to be sent is specified on the left hand side of this operator; the destination actor reference is specified on the right hand side of this operator. Consider the example shown in Listing 2.4. In this example, an actor has a reference to an instance of *Stack*. After this actor receives a *start* message, it creates a Stack and then initializes it by sending it an *init* message. When this actor receives the *replaceTopOfStack* message, it sends a *pop* message followed by a *push* message to the instance of Stack. In both cases, the destination actor reference (*s*) is specified explicitly.

Variations on the above syntax include replying to a message and forwarding a message. Replying to the sender of a message is a very common pattern; therefore, COOL provides a short hand for this, as illustrated in the example shown in Listing 2.5. In this

Listing 2.5: COOL Source for Replying to Messages

```
actor class Stack [atomic] {
  message top() {
    if (index == 0) {
      stackEmpty() => *;
    } else {
      stackTop(buffer[index - 1]) => *;
    }
  }
}
```

example, the top of the stack can be retrieved by sending the *top* message to an instance of *Stack*. When activated with this message, a *Stack* responds by sending a *stackTop* message if the stack is not empty and a *stackEmpty* message if the stack is empty. The operator '*' in this context represents a reference to the source actor who sent the message currently being dispatched. Use of the '*' operator is akin to a reply-to-sender. Whichever actor sends the *top* message to an instance of *Stack* must define *stackTop* and *stackEmpty* as part of its own messaging interface.

When a message is sent, the message itself contains references to both its source and destination actors. Forwarding a message allows the current actor to copy the message that it just received and then send it to a new destination, permitting the message to preserve the source actor reference. When forwarding a message, no message is specified to the left of the '⇒' operator. One possible use case for forwarding is where a generic service is offered, and upon requesting this service, a new actor is created each time to handle the service. Another possible use case for forwarding is where one actor serves as a proxy to another actor, forwarding all messages received to the actor that is actually responsible for handling the messages. This second use case is demonstrated by the example shown in Listing 2.6. In this example, a *StackProxy* actor is defined which behaves just like *Stack*, except that all messages received are forwarded to an instance of *Stack* that implements the actual stack behaviour.

In addition to sending messages, actors can store messages using COOL's *msg* data type. The syntax of message variables is illustrated in Listing 2.7. An example where this might be useful is a service for providing reliable message transfer between two actors.

Listing 2.6: COOL Source for A Stack Proxy

```
actor class StackProxy {
  Stack s;

  message init(int el) {
    => s;
  }

  message push(int el) {
    => s;
  }

  message pop() {
    => s;
  }

  message top() {
    => s;
  }

  message stop() {
    => s;
  }
}
```

The actor performing this service can retain a copy of the message it is trying to send until it receives an acknowledgement that it has been delivered successfully. This is to avoid unnecessary copying across transmission. There are two ways to assign a message to a message variable. The first way (illustrated by message handler *foo*) is where the message is explicitly created via a *new* statement. The second way is to assign an existing message to a message variable, as in the case where a received message is saved (illustrated by message handler *bar*). Messages contained within message variables can be sent by specifying the message variable of the left hand side of '⇒' and specifying the destination actor on the right hand side (illustrated by message handler *fooOrBar*). Message variables utilize resources from Steam; assigning a message variable to *nil* releases resources.

Listing 2.7: COOL Source for Message Variables

```
actor class A {
  msg m = nil;

  message foo(int x) {
    m = new foo(x)
  }

  message bar(int x) is recvMsg{
    m = recvMsg;
  }

  message fooOrBar() {
    if (m != nil) {
      m => *;
      m = nil;
    }
  }
}
```

2.2.4 Other Data Types: Timers and Time

In addition to actors and messages, COOL provides the following primitive data types: booleans, integers, enumerated types, floating point numbers, strings, timers, and time. We omit discussion of the first five data types, as they are very much like their counterparts present in most imperative programming languages. Language support for timers and time is less conventional, thus discussion of timers and time will be the focus of this section.

In COOL, a timer is a special built-in active object. It is created with a *trigger* expression and it can be cancelled with a *discard* expression. Like timers defined by Steam, there are two types of timers in COOL: one-shot and periodic. When either a one-shot or periodic timer expires, a timeout message is delivered to the actor that triggered it. If the timer is a one-shot timer, it is automatically discarded prior to the timeout message being dispatched. If the timer is a periodic timer, it will continue to timeout at the given frequency until it is explicitly discarded.

Creation of timers is illustrated by the example shown in Listing 2.8. Sending a *createTimers* message to the actor defined in this example results in the creation of two timers. The first is a periodic timer that expires every 100 milliseconds. Each time this

Listing 2.8: COOL Source for Timers

```
extern prints(string);

Actor A {
  Timer t1;
  Timer t2;

  message createTimers() {
    t1 = trigger timeout2() every 100 msec;
    t2 = trigger timeout1() on 10 sec;
  }

  message timeout1() {
    discard t1;
  }

  message timeout2() {
    prints("[timeout],");
  }
}
```

timer expires, it prints a timeout message to the console via an external function. The second timer is a one-shot timer that expires after 10 seconds. When this one-shot timer expires, it cancels the periodic timer via the *discard* statement. Timer durations can be arbitrary integers, where the units for the timeout must be one of sec, msec, or μsec.[5]

In COOL, *time* is a built in type. *time* consists of a 64-bit unsigned integer. The value of *time* represents the number of microseconds since the COOL epoch, which is a platform dependant constant. On Linux, this constant represents January 1, 1970. On other machines, the epoch could represent the time at which the machine booted. The COOL constant *now* is used for retrieving the current time. Operations allowed on *time* include comparisons and integer arithmetic. The actor presented in Listing 2.9 illustrates how a simple stop watch could be modeled using COOL.

[5]Timer resolution provided by Steam is to the nearest 1 msec or 10 msec, depending on a compile time options. For the purposes of experimentation for this thesis, timer resolution is set to 10msec.

Listing 2.9: COOL Source for A Stopwatch Actor

```
actor class StopWatch {
  time start = 0;

  message start() {
    start = now;
  }

  message stop() {
    if (start != 0) {
      stopReply(now - start) => *;
      terminate;
    }
  }
}
```

2.2.5 Runtime Error Handling

During the course of an executing COOL/Steam application, a number of runtime errors may occur. Some errors may be fatal (such as calling an external function that performs an operation that results in a C runtime error). Such errors are beyond the scope of COOL/Steam. Other errors can be detected and handled by the runtime system. Examples of runtime errors handled by COOL/Steam include a message delivered to an actor that is unable to handle it or a message destined for a non-existent actor. The COOL-level mechanism for reporting such errors is via a *notify* expression. A *notify* expression sends a special "exception" message to the actor representing Mom. In general, the COOL generated code and the Steam runtime look after such error reporting.

2.3 Steam API

The API to Steam is written in the programming language C, which was chosen as the language for its efficiency and portability.

2.3.1 Actor Identifiers: PIDs

In Steam, actors are identified by PIDs. A PID is a 32 bit value that is globally unique with respect to a single-node or distributed Steam application. When an actor is created, it is assigned a unique PID. From the perspective of Steam, all that is needed in order to communicate with another actor is its PID. A PID is equivalent to an actor reference in COOL. PIDs can be freely passed around from actor to actor within the body of messages. PIDS can also be compared for equality. Two PIDs denote the same actor everywhere if they are equal in value.

Five PIDs are reserved by Steam and have a special meaning: *none*, *self*, *owner*, *mom*, and *engine*. The first four of these PIDs are identical to their actor reference counterparts in COOL. The fifth reserved PID, *engine*, has no counterpart in COOL. *engine* is the PID of the actor representing the Engine. Each node has a single Engine actor. All of these PIDs are accessible via function calls provided by the Steam API. Furthermore, Steam provides a function for testing actor equality, `ActorEqual()`.

2.3.2 Steam Messages

Every message in Steam has the following elements: source PID, destination PID, message identifier, and message body[6] . The source PID represents the actor who sends the message. The destination PID represents the target actor to which the message is being sent. The message identifier is an integer, used for distinguishing one message from another. When an message is dispatched to an actor, the code for handling the message must look at the message identifier to determine what behavior it should exhibit and how it should interpret the rest of the message content. The message body consists of an array of bytes. The size of this array is set to 48 by default, but can be changed if required by the application. How data gets transferred to and from the message body array is the responsibility of the (Steam) application programmer.

Messages are sent via the API function `vmsend()`. This method takes the source

[6]In the Steam source implementation, a Steam message is labeled AcubeMessage. This name was changed for the purposes of COOL.

PID, destination PID, and message identifier as parameters, and returns a pointer to an "empty" message buffer byte array, which is to be filled by the sender. The destination PID may be an actor on the local node or an actor on a remote node. Actors can send as many messages as they wish. Messages are not actually sent until the actor commits or terminates its current message invocation. If the actor aborts, for whatever reason, all messages that were queued up to be sent are dropped. If Steam runs out of message buffers, a call to `vmsend()` will return `NULL` instead of a pointer to an empty message buffer.

In addition to the sending of messages, there are a few other functions for dealing with messages: `vmforward()`, `vmnotify()`, `vmallocbuffer()`, and `vmdeallocbuffer()`.[7] The function `vmforward()` allows the current actor to copy the message that it just received and then forward it to a new destination, permitting the message to preserve the original source PID and message type, but with a different destination PID. The `vmnotify()` is used for exception reporting and is provided to support COOL's notify statement; The purpose of `vmnotify()` is for reporting an error to another actor, such as Mom, where the original message header (source, destination, message identifier) are the parameters embedded within the content of the message body. Messages sent via `vmnotify()` are always sent at the end of an invocation; Messages sent using `vmnotify()` cannot be aborted. The functions `vmallocbuffer()` and `vmdeallocbuffer()` are provided to support msg variables in COOL. The function `vmallocbuffer()` is used for acquiring a pointer to a Steam message so that it can be maintained in an actor's local state variables across invocations. Once an actor is finished with the message, it calls `vmdeallocbuffer()` to remove its reference. Both `vmallocbuffer()` and `vmdeallocbuffer()` are subject to atomicity; that is, after an abort, the actor and Steam runtime undo any state changes that may have occurred as a result of calling either of these functions.

[7]Steam also has a `vmpriosend()` for sending priority messages. This was seen as a way to provide priority handling for events such as interrupts. We avoid discussing use of this feature because it in many ways violates the programming model for Steam. It is not used by COOL nor have we found a need to use this function in the course of experimenting with Steam and COOL. This function is likely to be removed in future versions.

2.3.3 Actors

An actor definition consists of six components: a state size, a constructor function, a behavior function, a class identifier, actor attributes, and an initialization message identifier. Each of these components is described in turn below. To represent all actors in a Steam application, an actor class table must be specified at compile time. The actor class table consists of an array of structures, where each structure defines a different actor class. The fields of the structure correspond to the six components. The actor class table is indexed by the class identifier. When using COOL, the COOL compiler automatically looks after generating the actor class table.

An actor's state size is the size of the data structure that represents its local variables that are preserved across invocations. When an actor is created, memory of this size is allocated for storing its local state variables. This memory is referred to as its state variables. In COOL terminology, this is referred to as local state.

An actor's constructor function is only called when the actor is instantiated. The purpose of the constructor function is to initialize this state variables. The constructor function is called within the context of the created actor, so references to *self* and *owner* within the context of the constructor are valid.

The behavior function is called when the actor is activated. Only when an actor receives a message will an actor's behavior function be called. Two arguments are passed to the behavior function: the actor's state variables pointer and the message which caused the actor to be activated. When an actor is activated, control of execution is passed to that actor and the actor is free to modify its state memory. Like the constructor function, references to *self* and *owner* are valid within the context of the behavior function. When the behavior function returns, it must return an enumerated type signifying whether it has committed, terminated, or aborted .

Each actor definition must be associated with a class identifier. The class identifier is used for differentiating actor types. Actors instantiated from a common class identifier can be considered actors of the same type. Likewise, actors instantiated from different class identifiers can be considered actors of different types. The class identifier is a component

of the actor PID, so that given a PID, an actor can determine its class identifier. The default range for this number is between 3 and 254[8]. When using COOL, the COOL compiler looks after assigning class identifiers to each actor class.

As described earlier, an actor definition can be tagged with certain attributes. Attributes are a way of assigning certain properties to an actor. In addition to conveying information pertinent to Steam, attributes provide a hook for a language like COOL, to associate information with an actor definition. Steam itself is only aware of two attributes relating to the Steam application boot process: *autoboot* and *bootable*. Each actor with the *autoboot* attribute is automatically booted when the Engine calls `Steam_Autoboot()`. In addition to *autoboot* actors, an application can specify its application boot actor, which can be used to differentiate the set of actors that startup on a given node. The boot actor is created when the Engine calls `Steam_Application_Boot()`. Each node can have a different boot actor, but all actors that are capable of being a boot actor must be tagged with the *bootable* attribute. The *autoboot* actors and application boot actors are always the immediate descendents of the Engine actor in an actor ancestor tree.

The final field in the actor class table structure is the initialization message identifier. The initialization message identifier is the message identifier to be used when a default initialization message needs to be sent to an actor. When an actor is created, an initialization message is typically sent, which specifies parameters for configuring the newly created actor. Upon being activated with its initialization message, the actor can be considered started, where by it can start sending messages, creating actors, creating timers, etc.... Under certain circumstances, such as when the Engine spawns a *bootable* actor or all *autoboot* actors, an initialization message must be sent to start an actor, but the Engine actor has no way of knowing what message identifier to use for the initialization message. In such a circumstance, the owning actor can resort to sending a default initialization message with message identifier defined by this field.

To instantiate an actor at runtime, the `vmnew()` function is provided. This function takes the actor's class id as a parameter and returns a PID for the newly created actor.

[8]Class identifiers 0, 1, 2, and 255 are reserved for *none*, *mom*, *engine*, and timers, respectively

Like other API functions, `vmnew()` is subject to atomicity. The actor created by calling `vmnew()` is not actually created until the actor commits. If the Steam Engine fails to create an actor, *none* is returned.

Steam also provides a mechanism to create an actor on behalf of another actor, namely `Steam_Surrogate_Create_Actor()`. This so-called surrogate create actor can be used to implement remote actor creation, where the *owner* and *self* exist on different nodes.

2.3.4 Timers

Steam provides functions for creating and canceling timers. Timers are a special type of build in actor. Like Actors, timers are also identified via PIDs. Each timer has a unique PID. When a timer expires, a message is dispatched to the actor that created the timer. The timeout message sent upon expiration is like any other message an actor could sent. It has a source PID, which represents the timer's PID. It has a destination PID, which represents the actor that created the timer. It has a message identifier, which is what distinguishes a timeout message from other messages that the actor might receive. Finally, it has a message body, where arbitrary data can be passed within the content of a message.

A timer is created via a call to `vmtrigger()`. This function's parameters are a pointer to a PID, a timer type, a timer duration, the units for duration, and a message identifier. This function returns a pointer to a message buffer byte array, which, like the `vmsend()` function call, represents the message body for the timeout message. The pointer to the PID is how the caller learns the PID of the newly created timer. The timer type must be either one-shot or periodic. The timer duration is an integer where its interpretation depends on units for duration, which can be microseconds, milliseconds, or seconds. The message identifier is for the timeout message sent upon timer expiration. If `vmtrigger()` fails to create a timer, the timer PID passed back to the calling actor is *none*.

There are two other timer-related functions in Steam: `vmdiscard()` and `vmvalidtimer()`. `vmdiscard()` is called to cancel a timer.`vmdiscard()` can only be called by the actor that owns the timer. Furthermore, because timers can only interact with

the actor that created them, communicating their PIDs in the content of messages serves no purpose. `vmvalidtimer()` is called to determine whether a given timer is still valid or not. A valid timer is one that has been created and has not yet expired nor been cancelled.

Both `vmtrigger()` and `vmdiscard()` are subject to atomicity. If `vmtrigger()` is called during an invocation and then the invocation aborts, the timer is not created. Likewise, if `vmdiscard()` is called during an invocation and then the invocation aborts, the timer is not cancelled. `vmvalidtimer()` also has interesting semantics with regard to atomicity. If a timer is cancelled via `vmdiscard()`, it remains a valid timer until the invocation commits.

2.3.5 Atomicity of State Variables

As described above, most operations that can be performed on the Steam runtime system are subject to atomicity. If an invocation aborts, the actions that would otherwise be carried out as a result of an operation are undone. However, before an actor aborts an invocation, it may have had the opportunity to modify its state variables. Without the ability to also restore its state variables to their original values, aborting an invocation could result in logic errors. For example, consider an actor who creates another actor during an invocation. Here, the act of creating another actor returns its PID, regardless of whether the invocation is committed or aborted. If the actor stores this value as part of its state variables and then aborts its transaction, its state variables would contain a reference to an actor that was never actually created. To address atomicity for state variables, the actor subsystem provides an interface for backing up the existing state, and then restoring it if the invocation aborts. Calling `vmcopystate()` at the start of an invocation will make a backup copy of the calling actor's state variables. Calling `vmrollback()` at the end of an invocation that is about to abort will restore that actor's state variables to its original state before the invocation.

2.3.6 Steam Configuration and Management Functions

The startup of the Steam runtime system is triggered by a call to `Steam_Start()`. Once this function is called, it does not return until the runtime is shutdown via a call to `Steam_Shutdown()`. An ungraceful shutdown of Steam is also possible via a call to `Steam_Die()`, where a call to the C-runtime system exit() function is made.

Steam is highly configurable. Before `Steam_Start()` is called, Steam must first be configured via a call to `Steam_Initialize()`. Steam configuration is based upon the steam configuration structure passed to this function. This structure defines a number of constants and parameters used throughout the course of the application. Information such as the actor class table is specified via this structure. Table 2.2 summarizes the fields of this structure.

2.3.7 Abstraction for Time

In addition to actor, timer, and messaging functionality, Steam also provides a portable abstraction for time, *ptime*. Although this abstraction need not be considered part of Steam per se, it is relied upon by Steam (and COOL). Steam provides the functions `T()` and `Now()` for accessing time. `T()` returns the current time (stored in a 64 bit integer). `Now()` returns the time at which an actor is activated (also in microseconds), which remains the same for the duration of the actor invocation.

2.4 Booting

As mentioned earlier, there are two kinds of Steam-based applications: single-node and multi-node. In a single-node application, all actors reside on the one node. Here, the one node has both a Mom actor and an Engine actor. In a multi-node application, actors can reside on different nodes. Here, each node has an Engine actor, but only one node has a Mom actor. This section provides examples showing how a Steam application boots, addressing both single-node and multi-node cases. Following a description of the boot

Table 2.2: Steam Configuration Parameters

Field	Description
Number of nodes	The number of nodes for which Steam should allocate resources.
Number of actors	The number of actors for which Steam should allocate resources.
Number of timers	The number of timers for which Steam should allocate resources.
Number of Local Messages	The number of message buffers that Steam should allocate for local messages sent between actors on the same node.
Number of Inbound Messages	The number of messages that Steam should allocate for messages received from remote nodes.
Number of Outbound Messages	The number of messages that Steam should allocate for messages to be sent to remote nodes.
Size of Messages	The number of bytes of the largest message used by actors.
State Size	The maximum state variables memory size for all actors.
Timer check frequency	A weighting for how frequently the main Steam dispatch loop should check for expired timers.
Inbound check frequency	A weighting for how frequently the main Steam dispatch loop should check for inbound messages via the transport.
Outbound check frequency	A weighting for how frequently the main Steam dispatch loop should check for outbound messages awaiting transmission.
Work request list size	This number represents the number of atomic operations permitted per actor invocation.
Is Mom	A Boolean which is true if this node is where Mom resides, false otherwise.
Is Distributed	A Boolean which is true if the application is distributed, false otherwise.
Actor Class Table	A pointer to the actor class table array.
Actor Class Table size	The number of elements in the actor class table array.
Boot class identifier	The class identifier for the application boot class for the current node.
Mom IP Port	The UDP Port number for node where Mom resides.
Mom IP Address	The IP Address of node where Mom resides.
Debug/Logging flags	Flags used for debugging Steam applications.

process, an example showing how actors discover other actors is presented.

In the following discussion, message sequence charts are used to describe interactions between the different agents in an application. The rectangles at the top of each chart represent agents. The dotted vertical lines extending down from the agents represent a timeline. The diagrams contain three types of agents: actors, the Steam runtime system for a given node, and the startup (`main()`) function for a given node. The startup function and the Steam runtime system are included as agents, since they play an active role during the boot process. Within the diagrams, agents can be considered concurrent. Activations (vertical rectangles) are used to show when an agent is active. An activation change from agent to agent is analogous to a context switch. Horizontal lines with a half arrow represent asynchronous operations, such as creation of an actor or sending an asynchronous message. Horizontal lines with a solid arrow represent synchronous operations, such as a procedure call to or from the Steam API. Synchronous operations are shown in order to illustrate the asynchronous, order preserving nature of Steam. For example, sending a message from one actor to another cannot activate the destination actor until it is invoked synchronously by its Steam runtime.

2.4.1 Single Node Boot Process

Figure 2.5 illustrates how a single-node Steam application boots. When the C startup function calls `Steam_Initialize()`, the Steam runtime system itself initializes, which results in the assignment of its node identifier. After initialization, Steam can be started by calling `Steam_Start()`. Once `Steam_Start()` has been called, the activation switches from the startup function to the steam runtime. `Steam_Start()` does not return unless Steam is requested to shutdown. `Steam_Start()` is responsible for starting the Steam runtime and for creating the node's Engine actor. Likewise, for the single-node case, the Mom actor is also created. Both of these actors are sent an *init* message which results in each of them being activated in turn. When the *init* message is dispatched to the Engine, the Engine initializes itself and then sends an *alive* message to Mom. When the *init* message is dispatched to Mom, Mom simply initializes itself. When an *alive* message is dispatched to Mom, Mom

responds by sending an *alive_ack* to the Engine. Once the *alive_ack* message is dispatched to the Engine, the Engine can proceed to startup application level actors. Starting up other actors is achieved by calling `Steam_Autoboot()` and `Steam_Application_Boot()`, which accesses the Steam configuration and the actor class table, looking for actors to create. For each actor created, it is sent a default initialization message, as defined within the actor class table. Once the newly created application-level actors are activated to handle the initialization message, the Steam application boot process is considered complete.

2.4.2 Multi-Node Boot Process

When discussing the booting of a multi-node Steam application, two scenarios must be considered. The first scenario is where the given node is the node that also hosts Mom. The second scenario is where Mom is hosted on a node remote to the given node. The first scenario is very similar to that presented in Figure 2.5. In fact, the only difference is internal to the Steam runtime system, where the act of initializing Steam results in its transport being initialized. The node identifier assigned to the given node must be the same as that associated with Mom. Apart from this subtle difference, the boot process proceeds in a manner identical to that outlined above for the single-node case.

Figure 2.6 illustrates how a node can be booted when Mom is remote. In this scenario, when `Steam_Initialize()` is called, the Steam transport is initialized, which results in the local node being assigned a node number. Once the local node has been assigned a unique node identifier, Steam can be started up in a manner similar to the previous cases. The main difference however, is that the local runtime does not create a Mom actor and that all communications with Mom take place via Steam's internode transport. Like the previous cases, the Engine waits for a response to its alive message before creating application level actors. Unlike the previous cases, overlapping activations are possible, since the local and remote Steam runtimes each represent a separate thread of control.

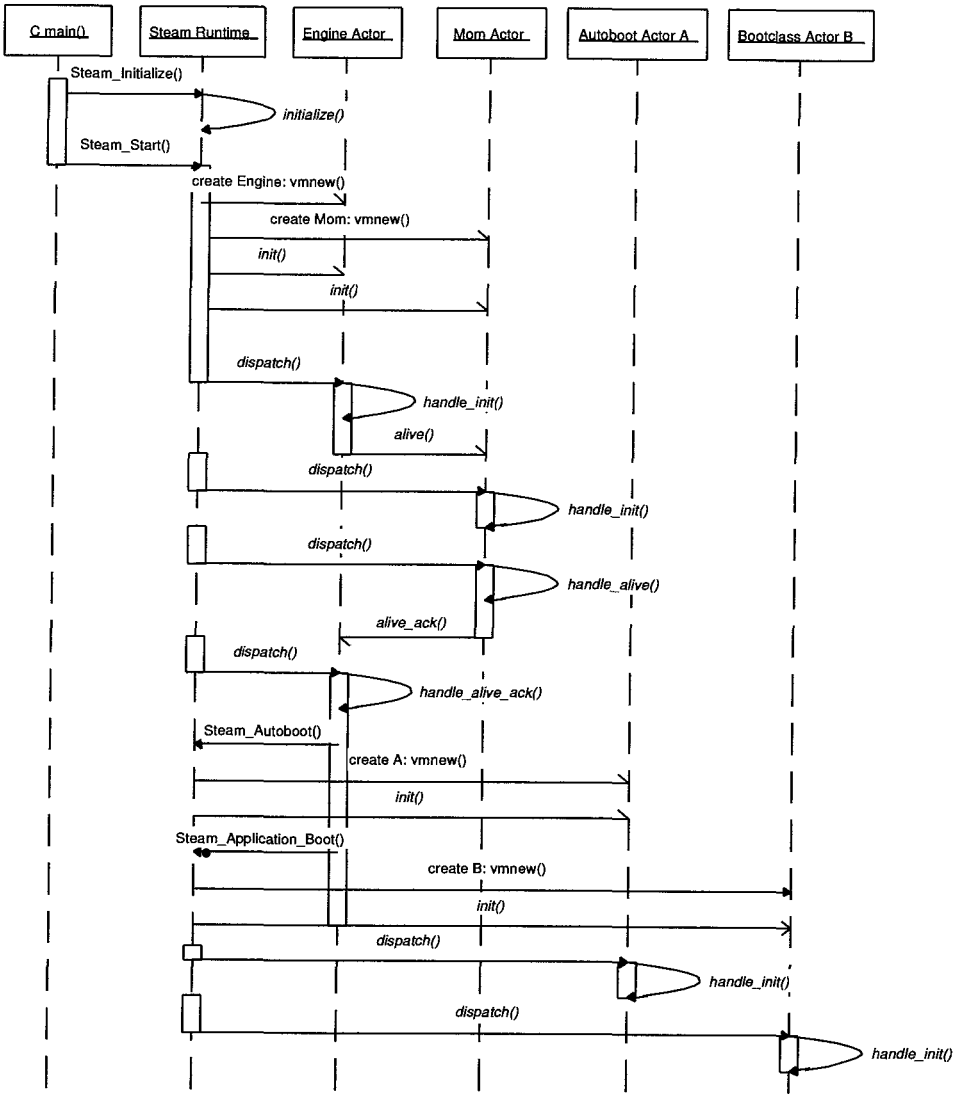

Figure 2.5: Single Node Boot Process

2.5 Discovery

After a Steam node successfully boots according to the scenario described above, the next objective is for actors to discover each other, assuming they do not already know of each other due to a common ancestry tree. To accomplish discovery, Mom takes on the role of

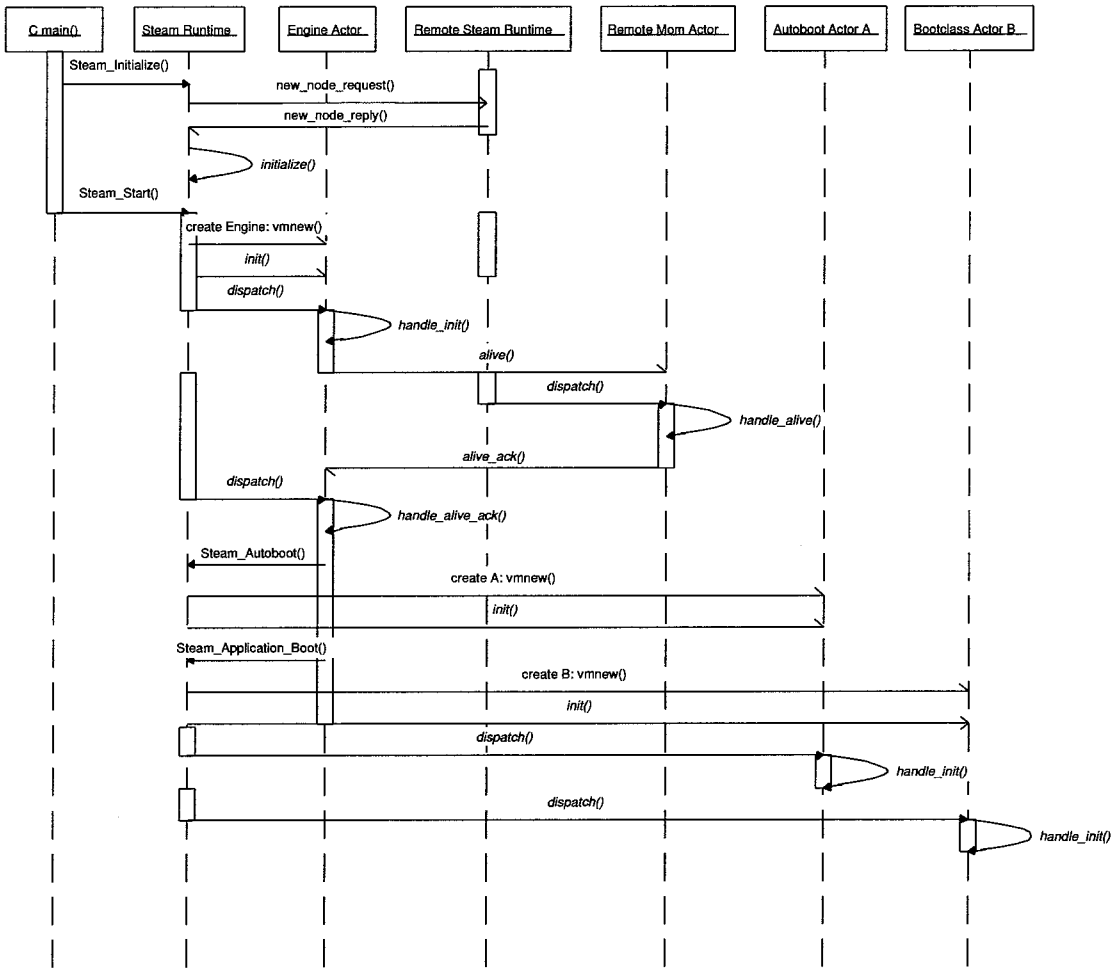

Figure 2.6: Multi Node Boot Process where Mom is Remote

the name server, providing the means by which to register and lookup services. Figure 2.7 shows a scenario, where two actors A and B discover each other via Mom. In this scenario, actor A resides on node X, Mom resides on node Y, and actor B resides on node Z. After A is activated with its initialization message, it registers itself with Mom, as providing a service S. Likewise, after B is activated with its initialization message, it sends a *lookup* message to Mom, requesting the PID of the actor providing service S. After Mom replies to the lookup, actor B learns the PID of A, such that B can send a message to A.

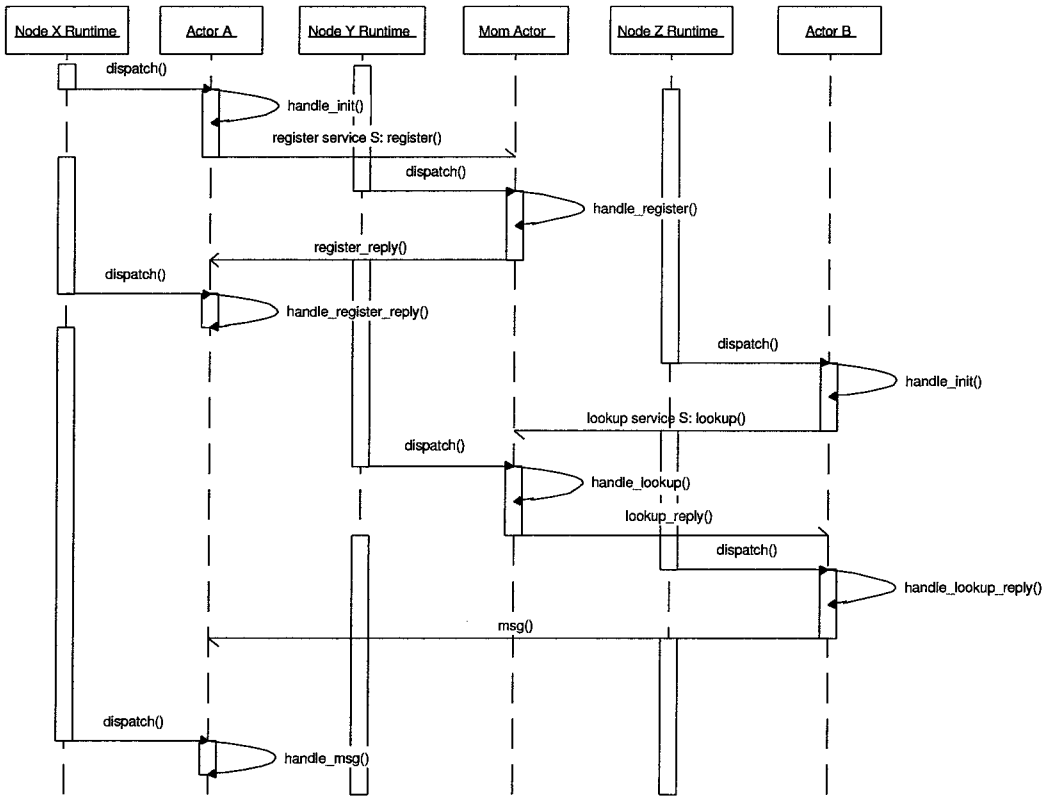

Figure 2.7: Actor Discovery via Mom

2.6 Application Programmer Issues

Steam has been presented as an API that can be used directly to build Steam-based applications. While this is accurate, it is important to note there are some problematic issues associated with using Steam directly. To help illustrate these issues, we discuss how syntax and semantic checks performed by a high level language like COOL can be used to address such issues.

A significant disadvantage to using Steam directly is that many of the issues relating to type safety are left up to the programmer. Alternatively, a big advantage to using COOL over "raw" Steam is that COOL enforces type safety, relieving the programmer from having to worry about many of the more tedious and error prone tasks associated

with Steam programming. Ensuring type safety for messages and the handling of messages once dispatched, are two examples that illustrate problematic issues faced by the Steam programmer.

Steam does not provide much in terms of support for type safety for messages. Steam simply delivers messages; it does not provide an ability to check whether the recipient of a message is capable of handling that message. Furthermore, Steam does not dictate the message identifiers which an actor uses to distinguish the different messages it receives. COOL provides a compile time type checking facility, where by the COOL syntax for sending a message from actor to another can be checked against the set of all message identifiers that the destination actor is capable of handling. Sending a message to an actor that it is incapable of handling is a semantic error that can be reported at compile time. COOL also looks after the generation of unique message identifiers for each message an actor is capable of receiving, relieving the programmer of having to manually perform this task.

Like any raw message passing interface, Steam does not provide a marshalling/de-marshalling function. It is the responsibility of the programmer to marshall data into the content of a message to be sent and to unmarshall data from content of messages received. COOL performs compile time syntax and semantic checking to make sure the contents of a message being sent from the source actor are what the destination actor expects in terms of data types. Furthermore, COOL emits the code that marshals data into a message and that de-marshals data from a message.

In addition to type checking and marshalling/unmarshalling data, COOL also implements the supporting code that safely handles a message, simulating a method invocation. With only a single behavior function associated with an actor, all "methods" supported by an actor must be implemented in terms of that one behavior function. If using "raw" Steam, this would amount to performing a select/case statement upon the received message's message identifier, casting the received message's parameters to the appropriate data types, and then executing the appropriate "method" behavior subject to the message's contents and actor's state. If a message identifier is not handled by the given behavior function, a default

behavior must also be provided to report an error or discard the received message. When using Steam directly, it is the responsibility of the programmer to call functions to manage atomicity for state variables. When using COOL, the COOL's atomic attribute for actors can be utilized, which ensures the appropriate C code is emitted to manage atomicity for state variables. Without the code generated by a language like COOL, the programmer must take great care developing code to safely handle each message and formulate a suitable response.

Chapter 3

Architecture and Implementation

The focus of this chapter is to describe the internal architecture of Steam, detailing how the various aspects of Steam are implemented. We start by providing a brief overview of the runtime system architecture. Following this overview, the different subsystems of the Steam runtime are described in detail. The implementation of Steam described here can be considered a reference implementation for the COOL virtual machine[9].

3.1 Internal Architecture Overview

The Steam runtime system can be decomposed into a number of inter-dependent subsystems (see Figure 3.1). The actor subsystem (Section 3.2) provides the physical representation for actors. All actors on a given node are represented by data structures contained within this subsystem. The timer subsystem (Section 3.3) provides the physical representation of timers. Running timers are implemented in terms of timer wheels and a delta list. The messaging subsystem (Section 3.4) provides message lists and message buffers. Message lists store messages sent by local actors and messages received from remote actors. The transport subsystem (Section 3.5) works in concert with the messaging subsys-

[9]It is based upon the most current version of Steam used by COOL (version 250). Although the original Steam was written by Paul Wierenga, the version described here has been modified by Gordon O'Connell for the purposes of COOL research, such as support for logical clocks. Steam is a work in progress; like any real system, it is subject to maintenance and modification over time by people other than the original author.

tem. The transport subsystem provides means by which outbound messages destined for remote nodes are transmitted and inbound messages from remote nodes are received. The dispatcher subsystem (Section 3.6) implements the Steam kernel, which co-ordinates the overall activities of the Steam runtime system.

The dispatcher executes in a continuous loop, polling the timer, transport, and messaging subsystems looking for work. Work takes the form of updating timers, transmitting or receiving messages via the transport subsystem, and dispatching messages to actors. The act of dispatching a message to an actor can result in any number of the following actions: termination of the actor, creation of other actors, sending of messages to other actors on the same node, sending messages to actors on different nodes, creation of timers, and cancellation of timers. When any such action is performed, the timer, transport, and messaging subsystems may be updated, such that more work may be generated for the dispatch loop on its next iteration.

The dispatcher is also responsible for maintaining an intentions list, which provides book keeping for atomicity. Actions, such as actor creation, timer creation, and timer cancellation, etc... which may be requested synchronously by an actor during an invocation, are not actually carried out in the context of the actor. Instead, they are performed asynchronously in the context of the runtime system. It is the purpose of the intentions list to record such actions. Only after an actor commits do the requests recorded by the intentions list get serviced.

We now turn our attention to discussing each subsystem in greater detail.

3.2 Actor Subsystem

3.2.1 Goals

The design goals for the actor subsystem are as follows:

- provide implementation for actor creation and destruction.

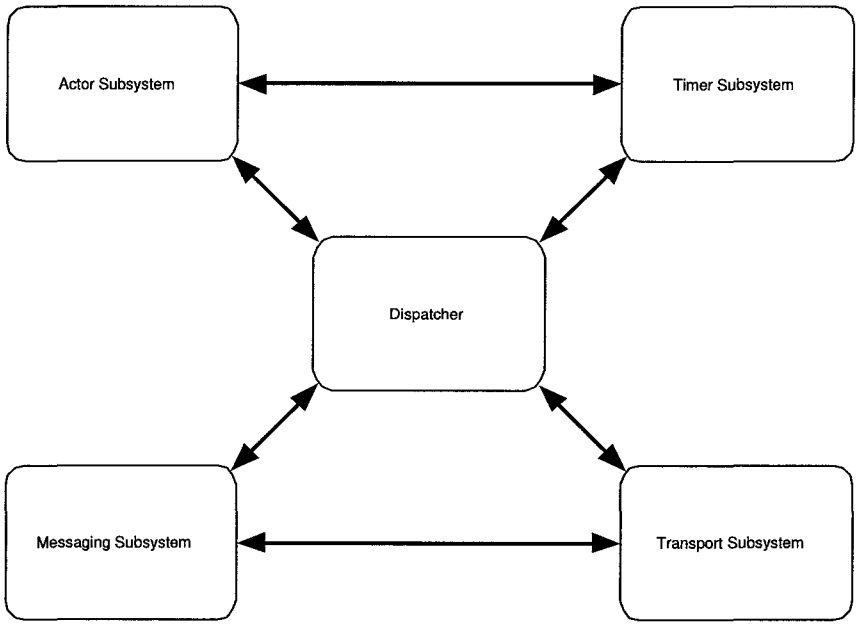

Figure 3.1: Internal Architecture of Steam

- provide an efficient representation of actors, where all memory required is preallocated and where actor lookup can be performed efficiently.
- provide a simple actor addressing scheme that is scalable and can promote location transparency.
- provide an efficient mechanism to support atomicity for actor creation and actor state changes.

3.2.2 Actor Representation

Internally, the Steam runtime maintains an actor descriptor record (ADR) for each actor that has been instantiated. The ADR is used by the Steam runtime for keeping track of all information relating to actors. The ADR contains a number of fields used by the Steam runtime to keep track of its actors. These fields are outlined in Table 3.1.

All memory for representing actors in Steam is pre-allocated. This memory consists of an array of ADRs and an array of memory blocks used for memory states. Configuration

Table 3.1: Fields of the Actor Descriptor Record

Field	Description
behavior	The function pointer, also referred to as *process*, called when a message is dispatched to the actor.
constructor	The function pointer that is called when the actor is first created. It initializes this actors' state variables.
activation time	A constant representing the time at which the actor was created.
state pointer	The pointer to its state variables.
state size	A constant representing the (fixed) memory size of an actor's state variables.
self	The PID of the given actor.
owner	The PID of the actor who created the given actor.
alive	A Boolean flag signifying whether the ADR represents an actor that is alive.

options passed into Steam during initialization determine the size of state variables and the number of actors that need to be supported.

In addition to the array of ADRs, the actor subsystem maintains a dead pool and a pending list. The dead pool is a singly linked list of ADRs not currently associated with any actor. The pending list is required in order to support atomicity. The pending list is a singly linked list of ADRs that have been allocated and associated with actors, but are not yet committed. The pending list can be considered an extension of the dead pool (see Figure 3.2). At the start of an invocation, the pending list is set to the dead pool (1). As actors are allocated, the pending list advances over ADRs that were previously considered members of the dead pool (2). If an invocation commits, the dead pool is set to the pending list (3a). If an invocation aborts, no action is required, since the pending list will be reset to the dead pool upon the next invocation (3b).

Actor creation takes place in two phases: allocation and construction. Actor allocation (calling `vmnew()`) consists of locating an unused ADR to represent the actor, assigning the actor a unique actor identifier, initializing its ADR, and inserting its ADR into the actor pending list. An actor remains in the pending list until the actor who has created it commits its invocation. Once committed, the construction phase of actor creation takes place. Here, within the context of the newly created actor, its constructor function is called, which

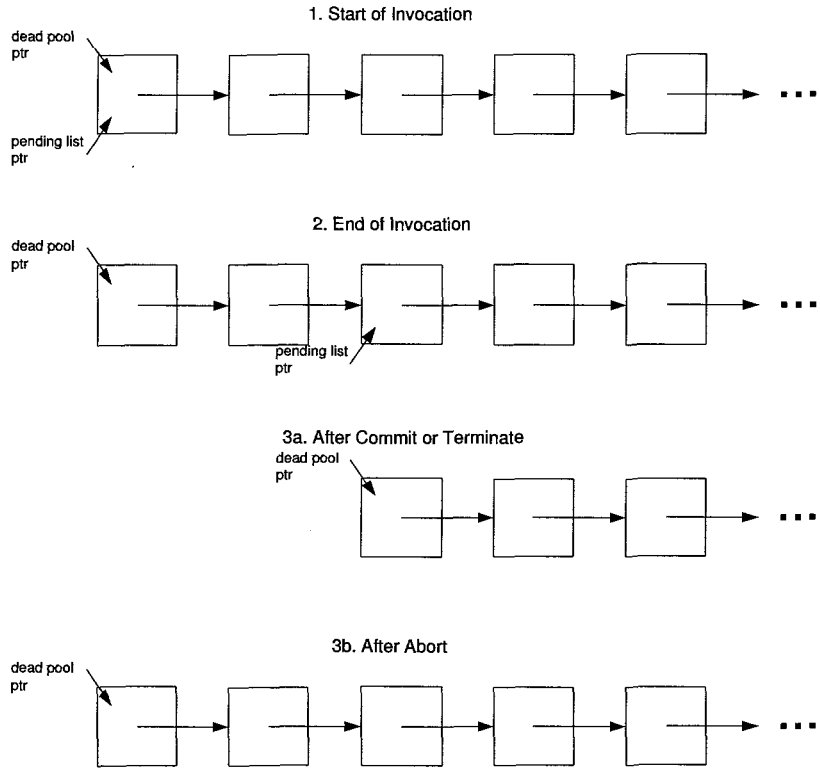

Figure 3.2: Actor Dead Pool List

initializes the actor's state variables.

When an actor terminates, the actor's ADR is returned to the dead pool, such that it may be reused when the next actor is created. As outlined above, when an actor aborts an invocation, all ADRs contained within the pending list also return to the dead pool. By pre-allocating and recycling the resources necessary for representing actors, actor creation and destruction is implemented efficiently, without the need for dynamic memory allocation.

3.2.3 Actor PIDs

The ADR for a given actor is located via its PID. The 32-bits of a PID are broken down as follows (see Figure 3.3). The upper 8 bits are reserved for the node identifier, permitting up to 256 nodes in a distributed Steam application. The next 8 bits represent the actor class

or type identifier. Up to 256 actor class identifiers can be supported. The lower 16 bits represent an instance number. One portion of the instance number represents an index into an array of ADRs, allowing for *O(1)* lookup of actors. The other portion of the instance number represents the generation number of a PID. The generation number is designed to provide some degree of uniqueness for PIDs, such that when actor resources are recycled, new actors are unlikely to be assigned PIDs that have been used before.

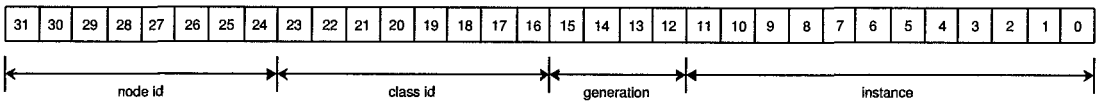

Figure 3.3: Actor PID Break Down

The actor identifier scheme used by Steam is quite flexible, and can scale up or down as the application demands. The size of the instance and generation portions of the PID is configurable when the Steam runtime is initialized. Furthermore, with only minor changes to the runtime system, the size of a PID can be increased or decreased, allowing the engine to support more or less nodes, actor classes, and instances.

3.2.4 Atomicity for Actors

The interface for backing up and restoring (local checkpointing) an actor's state variables is simple, yet efficient. Not all actors require atomic semantics. Therefore, a design decision was made to pass the responsibility of backing up and restoring state to the actor itself. Only those actors that require atomicity need call the interface for backing up (`vmcopystate()`) and restoring state (`vmrollback()`). Backing up an actor's state involves copying memory pointed to by its state variables pointer to a spare state variables pointer at the start of an invocation. If the actor commits (or terminates), there is no additional overhead introduced. If the actor aborts, the actor's state variables pointer is assigned to the spare state block, and the actor's original state memory becomes the spare state variables pointer for other actors to use. The overhead introduced by implementing atomicity for an actor's state is a memory copy proportional to the size of the actor's state memory and one pointer assignment if

the invocation is aborted.

As described earlier, Steam's programming model is highly asynchronous. The actual construction of an actor does not take place until the current invocation completes. Construction of actors is delayed so that if the invocation is aborted, actors are not unnecessarily constructed and then terminated. In addition to inserting the allocated actor into the pending list, each actor that is to be constructed after the invocation commits, is recorded in the dispatcher's intentions list. Once the actor commits, the actor allocations recorded in the intentions list are carried out by the runtime system. Carrying out the intent of the intentions list is the responsibility of the dispatcher.

3.3 Timer Subsystem

3.3.1 Goals

- provide a simple representation for one-shot and periodic timers.
- provide operations for creation and cancellation of timers.
- provide an efficient representation of timers, where all memory required is preallocated and where timer lookup can be performed efficiently.
- provide a flexible timer implementation that is precise, supports a large number of timers, and supports a wide variation in timer durations.
- provide an efficient mechanism to support atomicity for timer creation and timer cancellation.

3.3.2 General Approach

One of the first decisions made when designing the timer subsystem was to address how timeout messages were to be dispatched to actors. There are basically two approaches to this problem. The first approach is where timeout messages are treated like other messages

sent between actors. When a timer expires, the timer subsystem can send an asynchronous timeout message to the actor that created the timer. The second approach is where the timer subsystem directly dispatches timeout messages to the actor. Early on in the development of Crisp, (Steam's predecessor) the first approach was followed, mainly for simplicity reasons. However, three problems were identified with the first approach. First, the act of sending a timeout message creates the potential for a race condition, where the actor may try to cancel the timer after the timer has expired, but before the timeout message is actually delivered. Second, the act of sending a timeout as a message introduces the potential for a timeout to be lost, if the message itself was lost. If a timeout is lost, it can be very difficult for an actor to recover, since timers are viewed as a general error handling mechanism for situations such as the loss of a message. The third problem had to do with the accuracy of timers, where sending a timeout message would queue the timeout. If there were lots of other messages in queue ahead of the timeout message, the timeout message would not get dispatched until all other messages were processed first, thus causing additional delays for the timeout message. To avoid these problems, the second approach is the approach taken by Steam.

3.3.3 Timer Representation

There are two data types used in the timer subsystem: timer list and timer descriptor record (TDR). A timer list is a generic doubly-linked list for storing TDRs. A TDR contains the bookkeeping information necessary for maintaining timers. The fields of a TDR are described in Table 3.2.

All memory for representing timers in Steam is pre-allocated. This memory consists of an array of TDRs dynamically allocated during Steam initialization. Each timer that is created is allocated a TDR and assigned a PID. Like actor PIDs, the instance portion of the timer PID is an index into this array of TDRs, providing *O(1)* lookup of timers. Once the array of TDRs is initialized, TDRs must always exist in one of several timer lists: *dead*, *pending*, *created*, *discarded*, *timer wheel*, or *delta*. The *dead*, *pending*, *created*, and *discarded* timer lists represent timers that are considered not running. The *timer wheel* and

Table 3.2: Fields of the Timer Descriptor Record

Field	Description
next, previous	Pointers for maintaining a doubly-linked list.
type	One-shot or periodic.
tid	Timer PID.
alive	A Boolean flag signifying whether the TDR represents a timer that is alive and that should send a timeout message upon expiry.
start	The time at which the timer was started (used internally for determining how precise the timer is).
where	An enumerated type signifying which timer list the given TDR is currently a member.
duration	At creation time, the length of time that timer represents.
units	An enumerated type identifying the units for duration.
origTicks	An array of ticks where each element represents the number of ticks of a given grain.
insertLocations	An array of indices, identifying locations where a TDR should be inserted into a timer wheel.
delta	Delta ticks for large timers that have duration too large to be inserted into timer wheel
msg	The timeout message to be sent to actor upon timer expiration. Contained within this message is the PID of the actor that created the timer. The source PID for this message is the timer's PID.

delta timer lists represent timers that are considered running. Before discussing the purpose of each list and how a TDR migrates from one list to another, we must first address how running timers are managed.

3.3.4 Timer Wheels and Delta List

In Steam, running timers are maintained primarily by three hierarchical timer wheels[10]. A delta list is also provided to address overflow, where a timer's duration is too large to be handled by the timer wheels. A timer wheel is like a clock. A timer wheel has 10 spokes, where a turn of a single spoke represents a tick on that wheel. The three timer wheels tick at different rates. The first timer wheel has a tick resolution of 10msec. The second timer wheel has a tick resolution of 100msec. The third timer wheel has a tick resolution of 1000msec. When the first timer wheel makes one complete revolution (ie: wraps around), 100msec will have elapsed causing the second timer wheel to advance by one tick. Likewise, when the second timer wheel makes one complete revolution, 1000msec will have elapsed, causing the third wheel to advance by one tick. When the third timer wheel makes one complete revolution, 10000msec will have elapsed. Rather than having a fourth timer wheel, the decision was taken to use a delta list to maintain timers with duration larger than 10000msec. Therefore, when the third timer makes one complete revolution, the delta list is updated by one (delta list) tick. In this manner, the timer wheel can represent any duration of time between 10msec and 10000msec, down to a resolution of 10msec. Any timers with duration larger than 10000msec are managed by the delta list.

A delta list is a priority timer list where by timers are inserted in order of increasing duration. The timer with the shortest duration (nearest to expiry) is always at the front; the timer with the longest duration (farthest from expiry) is always at the back. When a timer is inserted into the delta list, a delta value for the timer is computed. This delta value is the difference between its duration and the cumulative delta values for all timers ahead of it in the list. For example, say we have four timers *a*, *b*, *c* and *d* to be inserted, with durations of

[10]The Steam source provides a fourth timer wheel to represent 1msec, but it is unlikely that applications in Steam would require timers of this resolution. Inclusion of a 1msec timer wheel is not necessary in order to convey Steam's concept of timer wheels, therefore, it is intentionally excluded from the discussion.

a=20, *b=30*, *c=60*, and *d=40*. Let us assume that the delta list is initially empty (*[]*). The first timer would be assigned a delta of 20 (20 - 0) and be inserted at the front (*[a=20]*). The second timer would be assigned a delta of 10 (30 - 20), and be inserted at the end (*[a=20, b=10]*). The third timer would be assigned a delta of 30 (60 - (20 + 10)) and be inserted at the end (*[a=20, b=10, c=30]*). The fourth timer would be assigned a delta of 10 (40 - (20 + 10)) and would be inserted in position after *b*, but before *c*. However, since it must be inserted before *c*, it must adjust the delta for *c* by subtracting its own delta from *c* (*[a=20, b=10, d=10, c=20]*). The primary advantage of a delta queue is that applying ticks to see if a timer has expired only involves checking the front of the delta queue, and decrementing its delta value. The biggest disadvantage of using a delta queue is that insertion can be expensive, as insertion of a timers is *O(n)*. An assumption made by Steam is that applications are more likely to require many fine grained timers. Therefore, the timer subsystem was optimized for timers with duration of less than 10000msec. As a result, most timers will not need to be inserted into the delta list.

Steam associates a timer list with each spoke of a timer wheel. If you include the delta list, this means there are 31 timer lists maintained by the timer subsystem to manage all running timers. Every 10msec of elapsed time results in at least one wheel advancing by one tick. As a timer approaches it expiration time, it gradually migrates from a spoke in its current timer wheel to a spoke in its next timer wheel, where the next timer wheel always represents a finer tick resolution. Once a timer has migrated to the timer wheel with the finest resolution (10msec), it can migrate no further. It is from this 10msec timer wheel that all timers eventually expire.

When a tick occurs on a wheel, the timer list at the given spoke must be serviced. Servicing a spoke means that the timers contained in that spoke's timer list must migrate to a spoke in the next timer wheel. Servicing of the delta list means that the timers ready to migrate from the delta list must migrate to a spoke in the timer wheel with the largest tick resolution. If there is no next timer wheel for a timer to migrate to (because the tick occurred on the 10msec wheel), the timer expires and the actor that created the timer is activated with an appropriate timeout message.

Migration of a timer from the delta list to a timer wheel or from one timer wheel to another timer wheel may seem expensive. However, the locations (spokes) in the different wheels where the timer will eventually get inserted are pre-calculated at the time when the timer is first inserted into either the delta list or one of timer wheels. This calculation is based upon adding a vector representation of the timer's duration to a vector representation of the current time. Current time is represented by the current positions of the spokes in the timer wheels. To illustrate how this computation works, consider the following example.

Let us assume that during an actor invocation, an actor calls `vmtrigger()` with a timer duration of 1250msec. After the actor invocation commits, the duration is broken down into different grains, representing the number of ticks for each timer wheel. A timer with duration 1250msec gets broken down into 5 * 10msec ticks, 2 * 100msec ticks, and 1 * 1000msec ticks. This information gets stored in the as the vector *[1, 2, 5]*, referred to as *origTicks*. As the timer is about to be inserted into the delta list or a timer wheel, its future locations (spokes) for where it must be inserted are determined. The *origTicks* vector is added (modulo 10) to the vector representing the current time. For example, say the current spoke locations for the three timer wheels are *[2, 3, 3]*. When we add the *origTicks* vector to this vector representing current time, we get the vector *[3, 5, 8]*, known as the *insertLocations* vector. This *insertLocations* vector is used whenever the TDR is migrated between spokes on different timer wheels. From the *insertLocations* vector, we know that we must insert our example timer into the 1000msec timer wheel at spoke 3. When the current spoke location for the 1000msec timer wheel advances to spoke 3, this timer's TDR migrates from the 1000msec timer wheel to spoke 5 of the 100msec timer wheel. When the current spoke location for the 100msec timer wheel advances to spoke 5, this timer's TDR migrates from the 100msec timer wheel to spoke 8 of the 10msec timer wheel. When the current spoke location for the 10msec timer wheel advances to spoke 8, it is time for this timer to expire and dispatch a timeout message to the actor who created it.

Listing 3.1: Pseudo-code for Timer Processing

```
tick:
    ticks = elapsed time in msec since last time

    ticks = ticks / 10

    while ticks > 0:

        apply tick to 10msec timer wheel

        if 10msec timer wheel wrapped:
            apply tick to 100msec timer wheel

            if 100msec timer wheel wrapped:
                apply tick to 1000msec timer wheel

                if 1000msec timer wheel wrapped:
                    apply tick to delta list
                    service delta list

                service current spoke of 1000msec timer wheel

            service current spoke of 100msec timer wheel

        service current spoke of 10msec timer wheel

        ticks = ticks - 1

    Insert any created timers into delta list or timer wheel
```

3.3.5 Tick Processing

The dispatcher subsystem periodically calls upon the timer subsystem to process any timers that may be running. When this occurs, the timer subsystem determines the amount of time that has elapsed since the last time the timer subsystem was invoked. For every 10 msec that has elapsed, the 10msec timer wheel must be advanced one tick. For each tick that is applied to the 10msec timer wheel, any potential changes to the other timer wheels or delta list must be performed before a subsequent 10msec tick is applied to the 10msec timer wheel. This procedure is outlined in Listing 3.1.

The act of servicing a delta list or the current spoke of the 100msec timer wheel or 1000msec timer wheel can only result in timers migrating to other timer wheels. The act of servicing the current spoke of the 10msec timer wheel can result in actors actually being

activated with timeout messages. The order implied by the pseudo code is important so that it is possible for a timer to migrate across more than one wheel in the course of processing a single 10msec tick. Also of importance is to refrain from inserting newly created timers into timer wheels until all tick processing is finished.

3.3.6 Timer Migration

With an understanding of the *timer wheel* and *delta* list mechanisms, we can now proceed to discuss how TDRs migrate between the various timer lists maintained by the timer subsystem. As stated above, TDRs must always exist in one of several timer lists: *dead*, *pending*, *created*, *discarded*, *timer wheel*, or *delta*. Figure 3.4 illustrates TDR migration between different timer lists.

The *dead* and *pending* lists are similar to their counterparts from the actor subsystem. When Steam is initialized, all timers are inserted into the *dead* list. When an actor creates a timer by calling `vmtrigger()`, a TDR from the *dead* list is inserted into the *pending* list. If the *dead* list is empty, `vmtrigger()` returns a PID of None for the timer. The timer subsystem's *pending* list and *dead* list function very similar to the actor subsystem's *pending* list and *dead* pool shown in Figure 3.2. Upon an invocation, the timer *pending* list is set to the *dead* list. As timers are allocated, the *pending* list advances over TDRs that were previously considered members of the *dead* list. If an invocation aborts, no action is required, since the *pending* list will be reset to the *dead* list upon the next invocation. If an invocation commits, the *dead* list is set to the *pending* list. When a one-shot timer has expired or when any timer (one-shot or periodic) has been *discarded*, it will eventually be re-inserted into the *dead* list. As timers are re-inserted into the *dead* list, the generation number of their PID is incremented, thus ensuring that the same TDR will have a different PID each time it is used.

The purpose of the *created* list is to be a temporary holding area for TDRs that are awaiting insertion into either the *delta* list or a timer wheel. Each operation to create or cancel a timer is recorded in the dispatch subsystem's intentions list. At the end of a successful invocation, each action contained within the intentions list is carried out. For

timers, this means that every TDR in the *pending* list gets added to the *created* list. A timer remains in the *created* list until it is safe to insert it into either the *delta* list or the timer wheel. Safety is an issue here because the current message dispatch could have been a result of a timer expiring. When the timer subsystem dispatches a timeout message to an actor, the timer subsystem could be in the middle of traversing a timer list contained within a *timer wheel*. Care must be taken to ensure that such a list is not corrupted by inserting new elements while it is being traversed. By buffering all *created* timers in a *created* list until the timer subsystem can safely insert timers into its *delta* list or *timer wheel*, integrity of such lists can be ensured.

Similar to the *created* list, the purpose of the *discarded* list is to ensure timer list integrity. When a timer is cancelled, it must eventually be inserted into the *dead* list. If a timer is cancelled while it exists in the *created* list, rather than immediately inserting the TDR into the *dead* list, the TDR is inserted into a *discarded* list. This is to prevent a scenario where TDRs are inserted into the *dead* list before the *dead* list is assigned to the *pending* list. In the commit scenario, after the *dead* list is assigned to the *pending* list, the *discarded* list is pre-pended to the *dead* list.

Once a TDR is inserted into either the *timer wheels* or the *delta* list, the TDR must migrate through the various *timer wheels* and expire. If it is a one-shot timer, expiration from the *timer wheel* results in the TDR being inserted into the *dead* list. If it is a periodic timer, expiration of the timer results in the TDR being re-inserted into the *created* list. If a timer is cancelled while it exists in either the *delta* list or *timer wheels*, the timer is marked as *dead* (not alive) and allowed to expire normally, with the exception that no timeout message be dispatched.

3.3.7 Implementation Issues

When it comes to canceling timers, Steam adopts a lazy philosophy. Early on in the development of Steam, a design decision was made to make the act of discarding a timer as fast as possible (from the perspective of the calling actor). Unlike timers in the *created* list, if a timer is considered running, canceling it does not force it to be removed from a *delta* list or

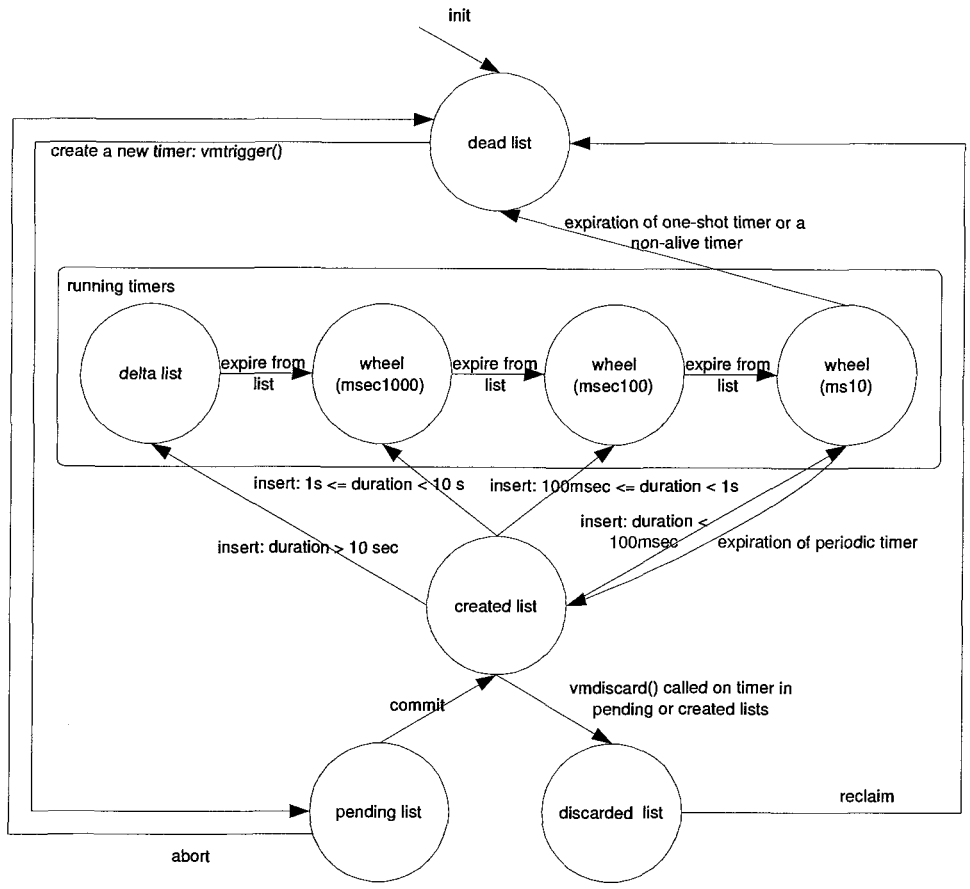

Figure 3.4: Timer Lists and TDR Migration

timer wheel until expiration occurs. This can cause a problem if a large number of timers are *created* and then cancelled. Although there should be ample timer resources available, this lazy cancellation of timers can cause artificial shortage of timers. The *discarded* timer list was invented to alleviate this problem. However, a better solution might have been to check the *alive* flag of a TDR whenever it migrates from one list to another. If a timer is not alive, there is no point migrating it from wheel to wheel, and the TDR could be recycled much quicker.

When Steam was first conceived, providing arbitrary timer durations was not a design goal. Early versions of Steam provided an interface to timers where a user would create a timer by specifying a grain for the timer and a multiple for the grain. For example, an

actor could create a 300msec timer by specifying 10msec as the grain and by specifying a multiple of 30. Alternatively, the actor could create a 300msec timer by specifying 100msec as the grain and by specifying a multiple of 3. This lead to a very simple implementation of timers, where timer wheels existed for each grain, but where there was no need to migrate timers from one wheel to another. Instead, such a timer would be checked at a frequency proportional to its grain. For the above case, if the grain was 10msec, the timer would be checked 10 times more frequently than if the grain was 100msec. The motivation for providing arbitrary timer durations came from COOL, where it seemed much more natural for a programmer to specify arbitrary timer duration than to specify a timer in terms of grains and multiples. The trade off of providing the convenience of arbitrary timer durations has perhaps made the timer subsystem more complicated than necessary.

The time reference used by the timer subsystem is provided by *ptime*, a portable timer library packaged with Steam. Reading the current time (via ptime), involves making a system call to `gettimeofday()`. Each time tick processing is performed, this system call is executed. If Steam is ported to a non-Linux system, the time services provided by ptime also need to be ported.

The final issue worth noting relates to the accuracy of periodic timers. Periodic timers are not truly periodic, but are subject to a slight drift over time. Instead periodic timers should be viewed more like one-shot timers that are automatically recreated upon expiration. Consider the following example. Let c represent the current time when a periodic timer is first created. Let d represent the duration of a periodic timer. When a periodic timer is created, it is set to expire at time $c + d$. However, because of how the timers are processed, let us say that the periodic timer actually expires at time $c + d + l$, where l is its latency factor. When the periodic timer is recreated, its next expiration time should be set to c + 2d. However, because Steam treats the periodic timer recreation more like a one-shot timer, its next expiration time turns out to be $c + d + l + d = c + 2d + l$. That is, over time, the periodic timer shifts a certain amount proportional to its latency factor. There are a few reasons for this latency factor. One reason is that Steam is busy executing a different part of the dispatch loop at the precise time when the timeout message should be dispatched.

Alternatively, if Steam is executing on a host operating system, such as Linux, the process running Steam could be pre-empted by another Linux process. If a timer is set to expire while the process running Steam is not currently scheduled to execute by the Linux kernel, the expiration of the timer will be delayed.

3.4 Messaging Subsytem

3.4.1 Goals

The design goals for the messaging subsystem are as follows:

- provide a simple and efficient representation of messages, where all memory required is pre-allocated.
- provide a messaging implementation that minimizes memory copying.
- provide a means by which an actor can save (allocate/de-allocate) messages.
- provide an efficient mechanism to support atomicity for sending messages.

3.4.2 Message Representation

Steam's messaging subsystem is what enables actors to send asynchronous messages to each other. To accomplish asynchronous messaging, the messaging subsystem uses three data types for representing messages: the message buffer, the message envelope, and the message list (see Figure 3.5). The message buffer is the memory that is used for representing a message. The message buffer is what is exported by the Steam API as a Steam message. It consists of a source PID, a destination PID, a message identifier, and a message body, as described in Section 2.3.2. The message envelope is a wrapper for a message buffer; each envelope contains a message buffer as well as a pointer to another envelope so that it can be linked into a list. A message list is a record containing two message envelope pointers, one for the head of a list and one for the tail. All buffering and queuing of messages within Steam is accomplished with these three data structures.

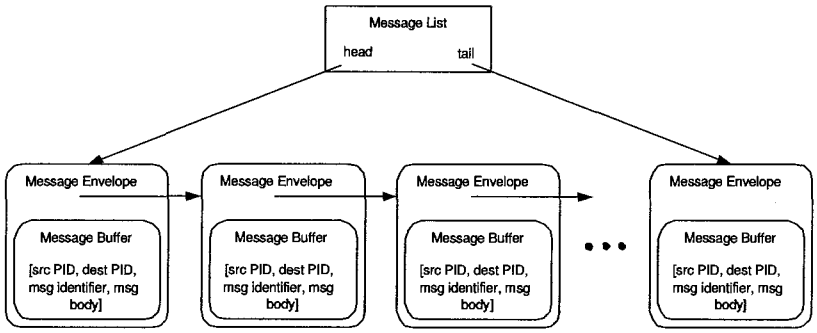

Figure 3.5: Message Buffer, Message Envelope, and Message List

Upon initialization of Steam, three different pools of message envelopes are allocated: local, inbound, and outbound. Local message envelopes are only used for messages sent between actors that reside on the same node. Inbound message envelopes are only used to store messages destined for actors on the local node, but received from remote nodes. Outbound message envelopes are only used to store messages destined for actors on a remote node. All messages sent between actors are be represented by message envelopes taken from one of these three pools. The size of each pool is configurable when Steam is initialized. Providing the ability to size each pool differently ensures that the exhausting of messages from one pool does not affect the other pools. For example, if all message buffers were allocated from a common pool, an influx of inbound messages could consume all message buffers such that no local or outbound messages could be sent. Keeping message envelopes separate from each prevents such situations from occurring.

For each message pool, a number of different message lists are used. Similar to how timers migrate between timer lists in the timer subsystem, message envelopes from a given pool migrate between message lists. We now turn our attention to describing the different message lists used for each pool, detailing how messages migrate between the different lists.

3.4.3 Message Migration

Message envelopes from the local message pool migrate between the following message lists: *localFree*, *localPending*, *localCommitted*, *allocatedPending*, and *allocatedCommitted* (see Figure 3.6). Upon initialization of the local pool, all message envelopes from this pool are inserted into the *localFree* list. A message envelope remains in the *localFree* list until either it is needed because an actor wishes to send a message to another actor on the same node or it is needed because an actor wishes to allocate a message buffer so that it can save a copy of a message. When an actor calls `vmsend()`, `vmforward()`, or `vmnotify()`, the destination PID is known such that the runtime can choose whether to use a message envelope from the local pool or outbound pool. If `vmsend()` or `vmforward()` is called, a message envelope is removed from the *localFree* list and appended to the *localPending* list. If `vmnotify()` is called, a message envelope is removed from the *localFree* list and appended to the *localCommitted* list. When an actor calls `vmallocbuffer()`, a message envelope is removed from the *localFree* list and inserted into the *allocatedPending* list. If the actor invocation aborts, all message envelopes in the *localPending* and *allocatedPending* lists are returned to the *localFree* list. If the actor invocation commits or terminates, all message envelopes in the *localPending* list are appended to the *localCommitted* list and all message envelopes in the *allocatedPending* list are appended to the *allocatedCommitted* list. It is from the *localCommitted* list that messages destined to local actors get dispatched. A message envelope remains in the *localCommitted* list until it is dispatched to its destination actor. Once a message from the *localCommitted* list is dispatched, it is considered delivered, and its envelope can be returned to the *localFree* list. When an actor is finished with an allocated message envelope, it is the responsibility of that actor to call `vmdeallocbuffer()`, which results in the message envelope returning to the *localFree* list after the invocation commits.

Message envelopes from the outbound message pool migrate between the following message lists: *outboundFree*, *outboundPending*, *outboundCommited*, and *outboundSent* (see Figure 3.7)[11] . Upon initialization of the outbound pool, all message envelopes from

[11]In the Steam source code, the *outboundCommitted* list is referred to as *outgoing* and the *outboundSent*

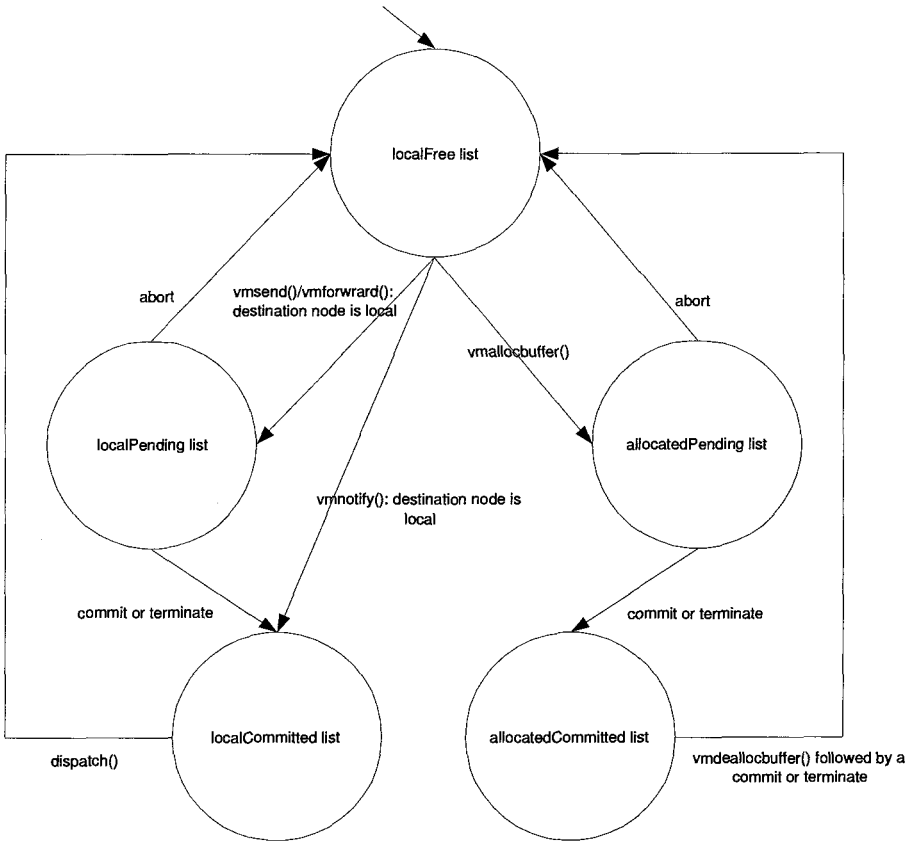

Figure 3.6: Local Message Pool - Message List Migration

this pool are inserted into the *outboundFree* list. A message envelope remains in the *outboundFree* list until it is required due to an actor calling `vmsend()`, `vmforward()`, or `vmnotify()`, where the specified destination PID is for a remote actor. When an actor calls `vmsend()` or `vmforward()`, a message envelope is removed from the *outboundFree* list and is appended to the *outboundPending* list. If an actor calls `vmnotify()`, a message envelope is removed from the *outboundFree* list and is appended to the *outboundCommited* list. If the actor invocation aborts, all message envelopes in the *outboundPending* list are returned to the *outboundFree* list. If the actor invocation commits or terminates, all message envelopes in the *outboundPending* list are appended to the *outboundCommited* list. It is from the *outboundCommited* list that messages destined to remote actors get transmitted

list is referred to as *sent*.

via the transport subsystem. Once the transport subsystem has transmitted the message, its message envelope gets placed into the *outboundSent* list. It will remain in the *outboundSent* list until the transport subsystem has finished processing, at which time the message envelope gets reinserted into the *outboundFree* list.

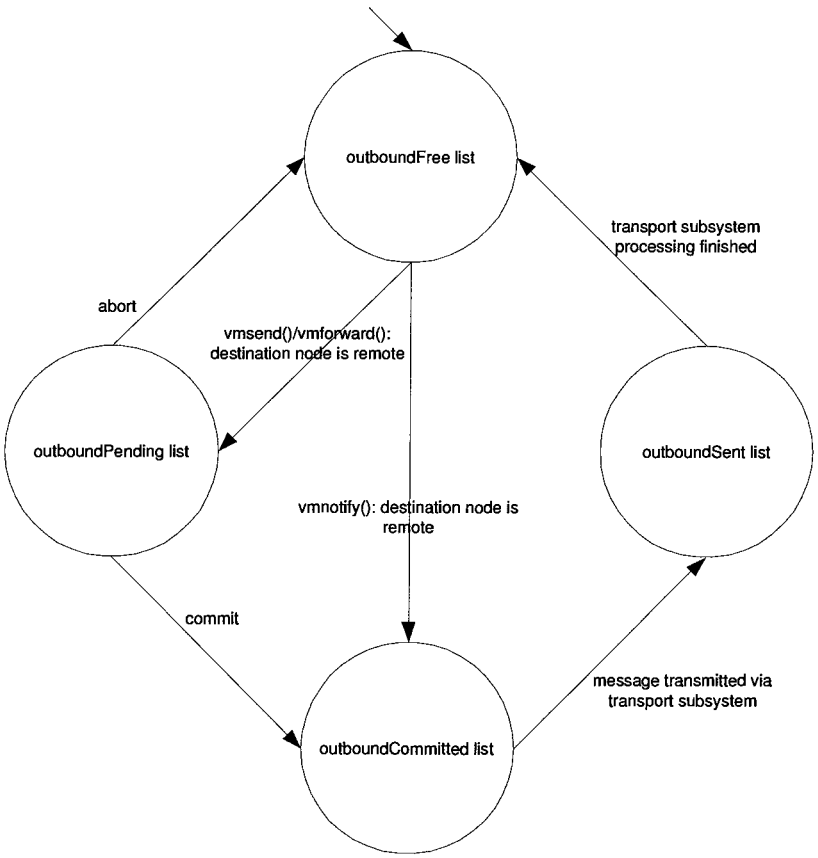

Figure 3.7: Outbound Message Pool - Message List Migration

Message envelopes from the inbound pool migrate between the *inboundFree* and *inboundReceived* lists (see Figure 3.8)[12] . Upon initialization of the inbound pool, all message envelopes from this pool are inserted into the *inboundFree* list. A message envelope remains in the *inboundFree* list until it is needed because the transport subsystem is about to receive a message from a remote node. Once a message has been received, its enve-

[12]In the Steam source code, the *inboundFree* list is referred to as *recvFree* and the *inboundReceived* list is referred to as *recv* list.

lope is inserted into the *inboundReceived* list, where it remains until it is dispatched to the destination actor.

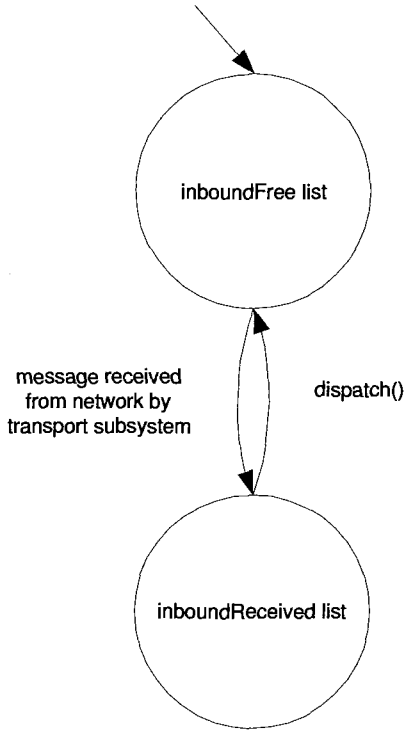

Figure 3.8: Inbound Message Pool - Message List Migration

3.4.4 Implementation Issues

In the process of migrating message envelopes between message lists, it is important to observe that message copying is minimized. Inserting message envelopes into lists and removing message envelopes from lists involve pointer assignment only. When the content of one list needs to be appended to another list, the whole list can be appended efficiently, without the need for appending each element individually. When an actor sends a message via `vmsend()`, a pointer to the message body is returned so that the actor can copy parameters directly into this buffer. Likewise, when an actor sends a message via `vmforward()` or `vmnotify()`, message content is copied once. When an actor is activated, it is passed

a pointer to the message buffer contained within its message envelope. When a message is received from a remote node by the transport subsystem, it is received directly into the message buffer, which in turn gets passed to the actor when the message is dispatched.

Order preservation for certain message lists is important. For the *localPending*, *localCommitted*, *outboundPending*, *outboundCommitted*, and *inboundReceived* lists, message envelopes must always be appended to preserve order. However, there are exceptions this rule. One can see that by calling `vmnotify()` to send a message, order preservation can be violated. Notifications are appended to either the *localCommitted* or *outboundCommitted* list, skipping over any messages contained in the *localPending* or *outboundPending* lists. As well, Steam does not provide ordering for messages from different pools. If three messages are sent by an actor, one local, one outbound, and one local, there is no guarantee that the outbound message will be dispatched (or even sent) before the second local message is dispatched. Scheduling of messages from different pools is the responsibility of the dispatcher subsystem.

The *forward anomaly* represents another situation where order of message delivery is compromised. For example, say we have three actors α, β and γ. Let α send a message *m1* to β and a message *m2* to γ. *m1* is dispatched to β before *m2* is dispatched to γ, and let us presume β forwards *m1* to γ (via `vmforward()`). Eventually, γ will be activated twice, first with *m2* and then with *m1*. From this example, it would appear that order preservation of messages is compromised by the act of forwarding a message, since γ has no way of knowing that *m1* was forwarded from β.

3.5 Transport Subsystem

3.5.1 Goals

The goals for the transport subsystem are as follows:

- provide a transmit/receive function that minimizes memory copying and ensures order of delivery of messages.

- provide an internode routing function for messages.
- provide a mechanism by which node identifiers can be assigned.
- provide a generic transport interface that could support different physical transports.

3.5.2 Topology and Protocol

When designing a transport for Steam, the first issue to be addressed is the network topology and underlying communication mechanism to be used. Steam employs UDP (over IP) as its underlying communication mechanism. Logically speaking, the topology of distributed Steam is a fully connected graph, in the sense that any node can communicate directly with any other node, in a peer to peer manner. In terms of the transport, it is not connected since UDP is a connectionless protocol. As depicted in Figure 3.9, each node has a single UDP socket that it uses to both send and receive messages.

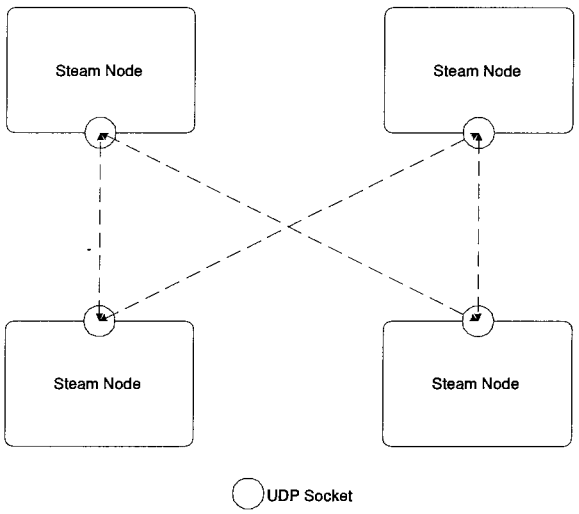

Figure 3.9: Steam Topology with UDP

The simplicity and efficiency of UDP are the reasons why it was chosen as the underlying communications mechanism. Since our target platform is Linux, we were limited to the most prevalent communication mechanisms on this platform: UDP and Transmission

Control Protocol (TCP, RFC 793), both over IP[13]. UDP is datagram oriented, connectionless, unreliable, and does not preserve order. On the other hand, TCP is connection oriented where each end point must adhere to the role of either client or server. Furthermore, TCP is stream-oriented, reliable, and order-preserving.

There are a number of advantages to using UDP instead of TCP. First, UDP is faster than TCP. TCP performs flow control and message retransmission. UDP just provides best effort delivery. Second, UDP is message-oriented. Steam messages are smaller than the UDP packet, therefore, Steam messages map easily into UDP packets. TCP is stream oriented, rather than message oriented, requiring the application to delineate message boundaries. Third, each node need only create one UDP socket to send and receive messages, making the model simple to implement. If TCP is used, a separate TCP connection would need to be established for each node in the system (see Figure 3.10). Furthermore, each side of the TCP communication would have had to take on the role of either server or client, making the initial setup of peer nodes in the fully connected topology more of a challenge.

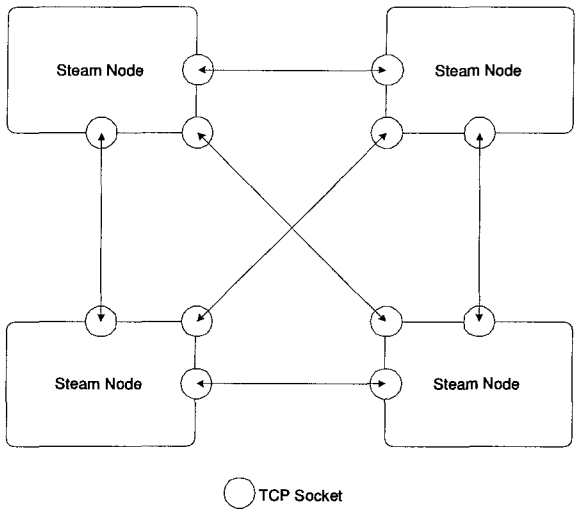

Figure 3.10: Steam Topology with TCP

There are two disadvantages to using UDP. The first disadvantage is that UDP is not order preserving. Steam requires order preservation of messages. The Steam transport

[13]Steam and COOL also run on Windows using UDP, but this thesis focuses on Linux.

implements logic to uphold this property. For each pair of nodes, the Steam transport maintains a sequence number. Each message sent is assigned a unique monotonically increasing sequence number. Each time the transport receives a message, this sequence number is examined. If the sequence number received is less than its last received sequence number, it drops the message. Otherwise, it sets its last received sequence number to the newly received sequence number. This simple method of dropping any out of order messages ensures that messages are always received in order (if they are indeed received). In addition to logic employed by Steam, the configuration of the network itself can influence whether UDP delivers messages out of order. For example, if the IP network hosting the Steam application is configured without loops, the possibility for out of order delivery of messages is reduced.

The second disadvantage to using UDP is that it is not reliable. No protocol seems to exist that is order preserving, asynchronous, and guaranteed[14] . There is not much the Steam transport can do to address reliability of UDP, unless a protocol for acknowledgements and retransmission of packets is developed on top of UDP. Steam makes the assumption that the underlying communication medium is highly reliable and that any loss of messages can be adequately handled by the Steam application. Even if Steam could address reliability of messages over UDP, there are still two circumstances under which Steam will drop messages. The first is the case described above, when a message is received out of order via UDP. The second is when an actor aborts, the message currently being processed by the actor is treated as a lost message. In light of all of this, it is reasonable to let the Steam application layer handle message loss.

3.5.3 Node Initialization and Routing Function

Closely related to issue of topology are the questions of where routing decisions are made and how to deal with nodes being added and removed from the system. In Section 2.5, we introduced the Steam application boot and actor discovery processes. Absent from

[14]Stream Control Transmission Protocol (SCTP, RFC 2960) does show some promise here. It is reliable, message oriented (asynchronous), and does provide guarantees for ordering within a stream, where a single SCTP session can support multiple streams.

Table 3.3: Fields in a Routing Table Record

Field	Description
sockAddr	A data structure representing IP address and port for remote node.
lookupTries	The number of node_lookup_request messages sent for this node.
lastIn	The sequence number of the last message received from this node.
nextOut	The sequence number for the next message sent to this node.
lookupInProcess	A boolean signifying whether a node_lookup_request message is currently outstanding.
alive	A boolean signifying whether this node has been communicated with yet

that discussion was the role played by the transport. The transport subsystem provides the means by which outbound messages destined for a remote node are transmitted and inbound messages sent from a remote node are received. As well, the Steam transport is responsible for the assignment of node identifiers when a node initializes.

Each node's transport subsystem contains a routing table. The routing table is indexed by the node identifier. Each record in the routing table contains information described in Table 3.3.

The routing table on the node hosting Mom can be considered the master routing table, because it is guaranteed to contain all nodes in a multi-node Steam application. When `Steam_Initialize()` is called in a multi-node Steam application, the Steam transport is initialized. Initialization of the Steam transport involves creating its UDP socket. If the node being initialized is not hosting Mom (which can be determined via the configuration information passed into `Steam_Initialize()`), the transport proceeds to initiate a dialog with the node hosting Mom. This dialog consists of sending a new_node_request message to the node hosting Mom. When the transport on the node hosting Mom receives this message, it adds the new node to its routing table, assigning it a unique node identifier, and then replies with a new_node_reply message. The new_node_reply message contains the node identifier for the new node. This sequence illustrated in Figure 3.11, which is similar to Figure 2.6, except that here the internal transport-level dialog is shown. Only after a

node identifier is assigned by the node hosting Mom, does `Steam_Initialize()` return. What takes place at the Steam transport level is transparent to the application level code.

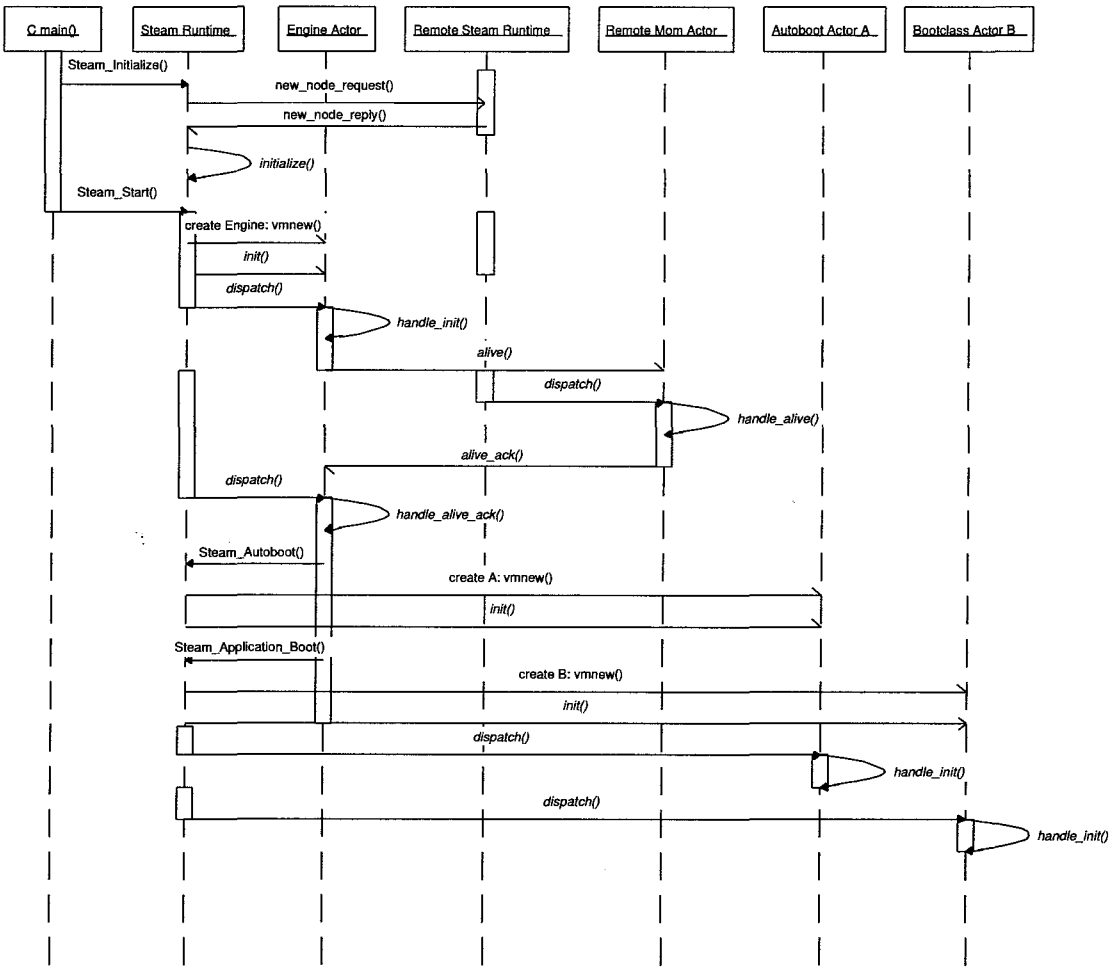

Figure 3.11: Multi-Node Steam Boot Process showing Transport Dialog

A routing table on a node that does not host Mom is updated in two situations. The first situation is when `Steam_Initialize()` is called, as outlined above. Here the routing table on the newly added node contains a single entry for the node hosting Mom. The second situation is when an actor on a node learns the PID for an actor on a remote node, where the remote node is not currently a member of the routing table. The situation is illustrated

in Figure 3.12, which is similar to Figure 2.7, except that here an internal transport-level dialog is shown. Transparent to the actors, a dialog takes place between the transport on node Z and the transport on the node hosting Mom (node Y). Here, a node_lookup_request is sent from node Z to node Y, where the node identifier for X is passed as a parameter. Upon receiving this request, the transport at node Y consults its routing table, and responds to node Z with the IP address and port for node X. Once this lookup completes, node Z updates its routing table, creates a UDP socket, and transmits the message on behalf of actor B to node X, where upon receipt, actor A gets activated.

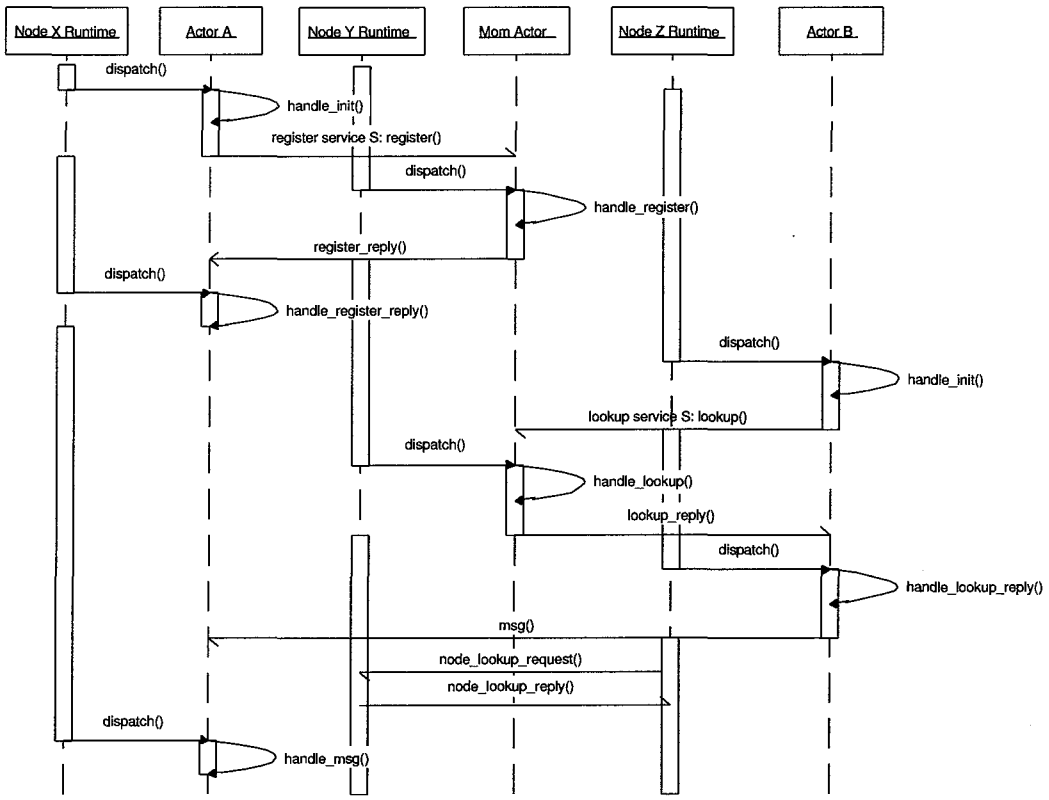

Figure 3.12: Actor Discovery via Mom showing Transport Dialog

3.5.4 Transport Mechanics

In addition to providing initialization and routing function as described above, the transport implements the mechanism for sending and receiving of messages. The transport subsystem can be further decomposed into an outbound portion and an inbound portion. The outbound portion is responsible for sending outbound messages. Likewise, the inbound portion is responsible for receiving inbound messages. The dispatcher subsystem periodically calls the transport subsystem to service either the inbound portion or the outbound portion. Discussion of the function of both the inbound and outbound portions of the transport is closely tied to the inbound and outbound message pools described in Section 3.4.3.

Sending of messages occurs in two phases. The first phase is where an actor invocation commits, and any *outboundPending* message envelopes are appended to the *outboundCommitted* list. The act of migrating outbound messages from a pending list to the committed list is accomplished within the context of the transport subsystem. The transport's internal interface provides a method for passing the message envelopes contained in the *outboundPending* list to the transport, where the transport proceeds to append them to its *outboundCommitted* list. At a later time, when the outbound portion of the transport gets an opportunity to be serviced, the transport proceeds to transmit messages contained in the *outboundCommitted* list. Transmission of a message involves writing the memory containing the Steam message to the UDP socket. There is no additional copying of memory in order to transmit messages. Once the transport has transmitted a message from the *outboundCommitted* list, it inserts the message envelope into the *outboundSent* list. At the end of servicing the outbound portion of the transport, all message envelopes contained within the *outboundSent* list get appended to the *outboundFree* list.

It is the goal of the outbound portion of the transport to transmit as many messages as possible while it is being serviced. However, if during the sending of a message, the transport encounters a destination node that is not in its routing table, it must stop (stall) transmitting actor messages until it either updates its routing table with an entry for that node or times out trying to update its routing table. Updating of its routing table is accomplished via the procedure outlined Figure 3.11. In general, when a Steam application is

running at steady state, the routing tables for all nodes will be the same, so that servicing of the outbound portion of the transport results in all messages in the *outboundCommitted* list being transmitted, without a stall.

The inbound portion of the transport operates in a manner orthogonal to the outbound portion. Periodically, the inbound portion of the transport is serviced, which checks to see if there are any messages to be received from UDP. For each message to be received, a message envelope from the *inboundFree* list is retrieved, and message data from the UDP socket is copied directly into the envelope's message buffer. For each message received, it is appended to the *inboundReceived* list. When servicing of the inbound portion of the transport completes, if the *inboundReceived* list is not empty, the first message in this list is returned to the dispatcher, where by the destination actor is invoked. Receiving of a message from a remote node involves only one memory copy, from the UDP socket to the message buffer contained within a message envelope.

3.5.5 Implementation Issues

The first thing to note about the transport subsystem is that the routing table is updated as necessary. The first message sent to a new node results in an expensive lookup that can stall transmission of all other outbound messages. However, once completed, the lookup procedure is not repeated. As an optimization, if a node receives a message from a node that is not currently contained within its routing table, the new node is added implicitly to the routing table.

The Steam transport is insulated behind a well defined interface. This was done so that a different communications mechanism could be used without affecting the rest of the Steam runtime or the application. For example, it should be possible to re-design the transport subsystem internals so that asynchronous transfer mode (ATM) could be used as the communications medium, without significantly impacting any other part of the system.

Part of the challenge of implementing the transport subsystem is to do it within the same thread shared by the rest of the system. Steam avoids blocking when sending and receiving via UDP by making all sockets non-blocking. Ideally, the interface to the com-

munications medium would be via direct memory access (DMA), where Steam's thread would only need to concern itself with manipulating message lists and the placement of buffers. An alternative design is to implement the transport subsystem within a second thread of control, so that this other thread can concern itself primarily with the task of transmitting and receiving messages. In such a scenario, the interface to the transport could stay the same, but suitable synchronization mechanisms would be required to provide safe updating of message lists.

The implementation of the Steam transport described here assumes that all nodes are homogenous with respect to sizes of primitive data types (int, bool, etc..) and with respect to byte order (Little Endian or Big Endian). Currently, neither COOL nor the internal inter-transport messages for maintaining routing tables are designed to handle non-homogenous nodes.

3.6 Dispatcher Subsytem

3.6.1 Goals

The goals for the dispatcher subsystem implementation are as follows:

- provide a simple run to completion model for actor invocation
- provide a mechanism for dispatching messages to actors
- ensure atomicity for actions performed during actor invocations
- schedule all Steam runtime activities to occur on a single thread of control
- provide a mechanism for merging local messages with those received from remote nodes

Listing 3.2: Pseudo-code for Dispatch Loop

```
Loop:

    If time to check timers:
        send tick to timer subsystem // which may dispatch timeout messages

    Else if time to service inbound transport:
        service inbound transport
        If there is an inbound message
            Call dispatch(message)

    Else if time to service outbound transport:
        service outbound transport

    Else if there are any local messages:
        Call dispatch(message)

    goto Loop
```

3.6.2 Dispatch Loop

The dispatcher subsystem implements the Steam kernel. The dispatcher is a single thread of control that executes in a continuous loop. Pseudo-code for the dispatch loop is outlined in Figure 3.2.

The dispatch loop is responsible for performing the following activities: sending *ticks* to the timer subsystem, servicing the inbound portion of the transport, servicing the outbound portion of the transport, and dispatching messages to actors from either the *inboundCommitted* or *localCommitted* message lists. Each time the dispatch loop is started, a check is made to determine whether it is time to perform one of these activities. Through the configuration structure passed to `Steam_Initialize()`, the developer can assign a weighting function to each of these activities. This weighting function predetermines the frequency by which the different activities get performed. For example, assigning a weighting function of 2 for timers, 4 for outbound messages, 4 for inbound messages, would result in the check for local messages being performed twice as often as the check for timers, and four times more often than servicing of either the inbound or outbound portions of the transport. In the dispatch loop's pseudo code, it is this weighting function that decides whether it is "time" to perform a certain activity or not.

3.6.3 Intentions List

In addition to the dispatch loop, the dispatcher subsystem is also responsible for the intentions list. All actions performed by an actor during its invocation get recorded in the intentions list. If the actor commits or terminates, its actions recorded in the intentions list are carried out by the dispatcher immediately after the actor invocation completes. If the actor aborts, its actions recorded in the intentions list are ignored by the dispatcher. The intentions list is represented by a ring buffer, where every action performed by an actor is recorded as an element of this buffer. Consider the following example (illustrated in Figure 3.13). At the start of an invocation, the intentions list is empty (its head and tail point to the same element as in 1). During an actor's invocation, it proceeds to create an actor (`vmnew()`), cancel a timer (`vmdiscard()`), and create a timer (`vmtrigger()`). At the end of the invocation, these three actions are recorded by the intentions list (2). If the actor commits or terminates, each action recorded in the intentions list is carried out, such that by the time the next actor is activated, the intentions list is empty again (3a). If the actor aborts, the tail of the intentions list is assigned to the head, with the effect that all actions performed by the actor are ignored (3b).

3.6.4 Dispatch Function

The actual dispatching of an actor and processing of the intentions list is performed by the dispatch function. The dispatch function is called by both the dispatcher loop and the timer subsystem when a timer has expired. The pseudo code for the dispatching function is outlined in Listing 3.3. When a message is dispatched to an actor, the dispatcher locates the target actor via the actor subsystem, and then invokes the actor. After an actor invocation completes, the dispatcher either performs work recorded in the intentions list, thus committing the contents all pending lists, or resets the intentions list and all pending lists.

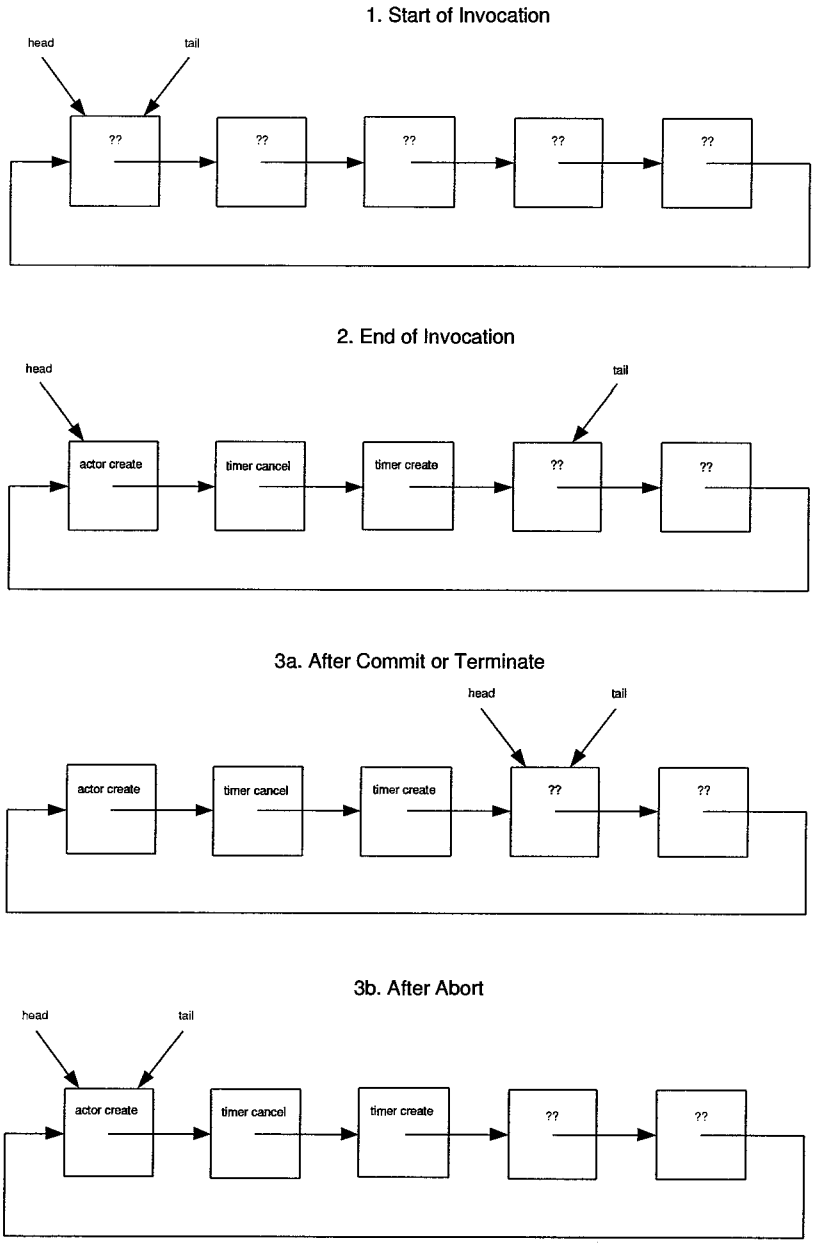

Figure 3.13: Intentions List Processing

3.6.5 Implementation Issues

The size of the intentions list is configurable at the time when Steam is initialized. If an actor performs so many actions that the intentions list's ring-buffer wraps around, the Steam

Listing 3.3: Pseudo-code for Dispatch Function

```
Function dispatch(message m):
  Determine current time (Now())

  Lookup actor identified by destination of m

  Invoke actor's behavior function

  If invocation returned commit or terminate:
    Service intentions list
    Commit pending lists
    If invocation returned terminate:
        Destroy actor

  Else if invocation returned abort:
    Reset intentions list
    Reset pending lists

  Return
```

runtime exits. Use of the intentions list is a runtime resource, like a call stack. If a process or a thread exceeds all the memory reserved for its call stack, it usually exits as well.

You will note that the dispatch loop shown in Listing 3.2 never yields. Steam was originally conceived to be the only application running on the processor. If there is no work for the dispatch subsystem to perform, the dispatcher loops continuously in a busy-loop. If this behaviour is viewed as being undesirable, it would be simple enough for the code to sleep or yield whenever the dispatch loop found no useful work to perform. Alternatively, a weighting function could be introduced such that the dispatch loop yields at a certain frequency relative to its other activities.

One of the biggest challenges of implementing the transport subsystem was to come up with a scheme by which to merge the stream of *inboundReceived* messages with the stream of *localCommitted* messages, so the dispatcher could dispatch messages from either stream. By providing a weighting function for each activity that the dispatch loop performs, the application programmer has some control over how frequently messages are dispatched from each stream. However, there is still a problem with this approach. Consider a scenario where the number of local messages (in the *localCommitted* list) is quite large and where servicing the inbound and outbound transports is quite frequent. Let us assume an actor α

sends two messages *m1* and *m2*, where *m1* is destined to local actor β and *m2* is destined to remote actor γ. Because of the backlog of messages in the *localCommitted* list, *m1* could be dispatched to actor γ first. If γ then sends or forwards a message *m3* to actor β, depending on the weighting function and backlog of *localCommitted* messages, it is possible *m3* to be dispatched to β before *m2* is dispatched to β. Though this scenario is unlikely, it is akin to the *forward anomaly* presented in Section 3.4.4.

Chapter 4

Examples and Performance Measurements

The purpose of this section is to provide some examples that demonstrate how Steam works and to provide empirical results showing its performance characteristics. The examples presented highlight the performance of various aspects of Steam: atomicity, actor context-switching, intranode messaging, internode messaging, and timers.

All code presented is either shown using pseudo code or actual COOL code. All examples were executed on some combination of three computers running the Redhat 9 distribution of Linux with version 2.4 of the Linux kernel. Two of the computers, referred to as *D1* and *D2*, are machines with dual-processors, running at a speed of 733MHz and with 512MB memory. The third computer, referred to as *S1*, is a single processor machine, running at 733MHz and with 512MB memory. Each computer has a 100Mbit ethernet network interface card, which is connected to an ethernet switch. All computers reside on the same IP subnet, but are not necessarily connected to the same switch. Furthermore, the network is used by a large number of other computers. Tests were run during non-busy hours when network activity was minimal, but the network cannot be considered a dedicated "Steam" network.

Listing 4.1: COOL Source for Sieve Method - Generator Actor

```
actor class Generator {
  int card = 2;
  Filter f = none;

  message INIT() {
    f = new Filter;
    INIT(card) => f;
    next() => self;
      commit;
  }

  message next() {
    card = card + 1;
    next(card) => f;
    next() => self;
    commit;
  }
}
```

4.1 Prime Number Generator

The purpose of the first example is to provide a sense of what a COOL/Steam application looks like and to measure the overhead of atomicity for state variables. The Sieve of Eratosthenes is a classic solution to the problem of generating prime numbers. For our first example, we show how Steam can be used to implement a Sieve method [21] for generating prime numbers.

The COOL code for our prime number generator is presented in Figure 4.1 and Figure 4.2. This program is an example of a single-node Steam application, where all communication occurs between actors on the same node. There are two classes of actors: *Generator* and *Filter*. The *Generator* is responsible for generating a sequence of monotonically increasing numbers starting at 2. A *Filter* is created for each prime number. The *Filter* is responsible for filtering out any number that is evenly divisible by its prime. If a number is not evenly divisible, it is forwarded to the next *Filter*. If there is no next *Filter*, the number must be a prime, and a new *Filter* is created. Steam supports up to 4096 actors, using this program, upto 4093[15] prime numbers can be computed before Steam runs out of ADRs.

[15]Why not 4096? One actor is used for each of Mom, Engine, and the generator, leaving 4096 - 3 = 4093.

Listing 4.2: COOL Source for Sieve Method - Filter Actor

```
actor class Filter {
  int prime = 0;
  Filter f = none;

  message INIT(int num) {
    prime = num;
    commit;
  }

  message next(int num) {
    if ((num % prime)==0) {
      // not a prime
      commit;
    } else {
      if (f == none) {
        // prime
        f = new Filter;
        if (f == none) {
          cool_exit();
        } else {
          INIT(num) => f;
        }
      } else {
        next(num) => f;
      }
    }
    commit;
  }
}
```

The performance of this application is summarized in Table 4.1. Six variants of the basic prime number generator were executed on machine *S1*. Each variant was executed 5 times. All variants are based on the code presented above. The differences stem from the size of state memory associated with each actor and whether the actors are atomic. An actor that is atomic makes calls to back up its state variables (`vmcopystate()`) at the start of each invocation. The size of memory for state variables can be one of default, default + 512 bytes, or default + 1024 bytes, where default is the 8 bytes used to represent an actor and an integer.

Table 4.1: Sieve Method - Performance Summary

Run Number	1	2	3	4	5	6
Number of local messages	9322940	9322940	9322940	9322940	9322940	9322940
Size of memory for state variables	Default	Default + 128 * 4 bytes	Default + 256 * 4 bytes	Default	Default + 128 * 4 bytes	Default + 256 * 4 bytes
Atomicity	No	No	No	Yes	Yes	Yes
Average execution time in seconds (stdev)	17.15034 (0.02959)	17.60393 (0.02435)	17.69052 (0.04111)	18.45402 (0.06564)	33.10521 (0.04895)	42.10061 (0.06549)

4.1.1 Observations

Measuring the duration of the program with such variables gives a sense of the cost of atomicity. The main point of this example is to show that the cost of providing atomicity for state variables is proportional to the size of an actor's state variables. This cost is graphically illustrated by Figure 4.1. This figure also underlies the importance of avoiding memory copies at all costs.

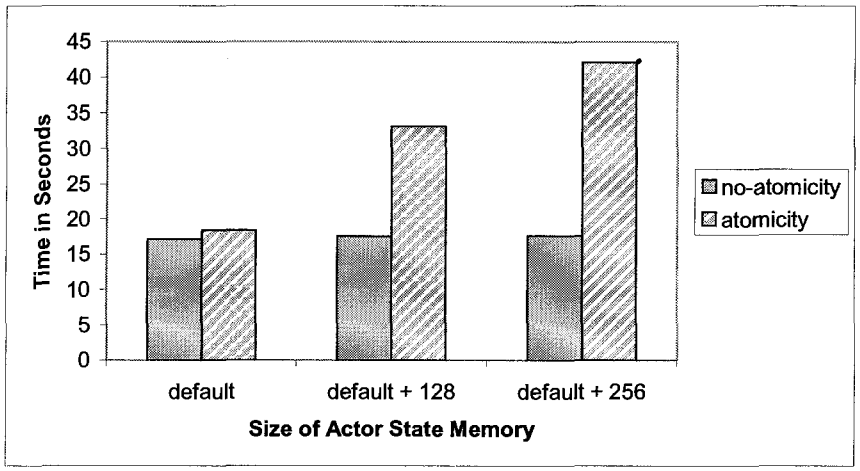

Figure 4.1: Sieve Method - The Cost of Atomicity

4.2 Inter-actor Context Switch Time

The purpose of the second example is to measure the time it takes for an actor to send a message to itself and then to be activated to handle that message. This is analogous to the context switching for threads in an environment that supports multithreading. To put Steam's performance into perspective, the inter-actor context switch time of Steam is compared with the context switching time for threads.

The COOL code for this example is outlined in Listing 4.3. The *Main* actor starts its "stopwatch" and then spawns the *Context* actor, initializing it with a counter of 100. The *Context* actor, in turn sends a message to itself, decrementing the counter each time it receives a message. When the counter reaches 0, the *Context* actor signals its owner to stop its "stopwatch".

As a comparison, Listing 4.4 shows the pseudo code for measuring context switching time in a multi-threaded environment. At the start of the thread, the "stopwatch" is started. Once per iteration of the main loop, the thread yields its control over the processor. After 100 iterations, the "stopwatch" is stopped.

Table 4.2 presents a summary of results for the two cases described above. These

Listing 4.3: COOL Source for Measuring Inter-Actor Context Switch Time

```
actor class Context {
  message run(int x) {
    if (x == 0) {
      done() => owner;
      terminate;
    } else {
      run(x-1) => self;
      commit;
    }
  }
}

actor class Main {
  message INIT() {
    Context c = new Context;
    start_stopwatch(); // external function
    run(100) => c;
    commit;
  }

  message done() {
    stop_stopwatch(); // external function
    cool_exit();
  }
}
```

Listing 4.4: Pseudo Code for Measuring Thread Context-Switching Time

```
thread:
    count = 100
    start_stopwatch();
    loop:
        count = count - 1
        if (count > 0):
            yield()
            goto loop
    stop_stopwatch()
```

Table 4.2: Inter-Actor and Thread Context Switch Times Summary

	Context Switching Time for Actors on *S1* in μsec (stdev)	**Context Switching Time for Threads on *S1* in μsec (stdev)**	**Context Switching Time for Actors on *D1* in μsec (stdev)**	**Context Switching Time for Threads on *D1* in μsec (stdev)**
1 actor/thread, 1000 iterations	1.71 (0.206)	0.691 (0.0260)	1.751 (0.0123)	0.855 (0.0152)
10 actors/threads, 1000 iterations	1.55 (0.0335)	0.910 (0.00242)	1.63 (0.0142)	1.24 (0.0550)
100 actors/threads, 1000 iterations	1.53 (0.0154)	2.32 (0.0393)	1.60 (0.0163)	3.17 (0.0200)
1000 actors/threads, 1000 iterations	1.53 (0.00726)	2.44 (0.00639)	1.60 (0.00649)	3.21 (0.0127)

Table 4.3: Resource Usage Summary For Actors and Threads

	Total Memory Usage for Actors in bytes	**Total Memory Usage for Threads in bytes**
1 actor/thread	40	1024
10 actors/threads	400	10240
100 actors/threads	4000	102400
1000 actors/threads	40000	1024000

programs were executed on both a single processor (*S1*) and dual-processor (*D1*) machines. Tests performed were subject to two variables: number of concurrent actors or threads and number of iterations. All times presented represent the number of μsec per context switch. For the actor cases, the time to create the actor is included in the measurement. For the multithreaded cases, the time to spawn the thread is not included in the measurement.

Table 4.3 presents a summary of memory used by each of the two cases. In the case of actors, memory usage is due to the actor descriptor record (40 bytes) and state memory associated with the actor (0 bytes). In the case of threads, memory usage is due to the the call stack that must be allocated for each thread.

4.2.1 Observations

Analysis of these results yields a number of interesting observations. First, the figures suggest that for a simple program that does no useful work, the context switching times for multithreaded programs are comparable to the inter-actor context switching time. For cases where there is only one thread, the Steam application is outperformed by its counterpart[16]. For all other cases, the Steam application exhibits slightly better performance. A second observation has to do with the variation in inter-actor context switching times versus variation in thread context switching times. For the actor case, regardless of the number of actors, the inter-actor context switching times remain quite constant. For the multithreaded case, the context switching time increases as the number of threads increases. Also noteworthy is that for both the Steam-based and multithreaded based programs, the single-processor machine outperforms the dual-processor.

It is reasonable here to conclude that actors are more scalable than threads. In addition to context switching for actors out performing its threading counterpart, the memory and system resources required to run the multi-threaded example was much greater than the requirements of actors. For each actor, one ADR plus the state memory to represent the actor was allocated. For each thread, a call stack (1 kilobyte) and program counter had to be maintained by the kernel. If real work was taking place, the call stack for threads would generally have to be much larger, on the order of tens of kilobytes. As well, for threads, each context switch resulted in transitioning from user space to kernel space, where the kernel could run its scheduler, and then back to user space. For actors, all context switches took place in user space.

[16]Although this may appear to be the case, about 0.5μsec of overhead is introduced by calling the `gettimeofday()` function once per message dispatch in order to provide the `Now()` function, as part of the Steam API. On a system where reading the current time does not entail a system call, the actor context switching time would be further reduced

Listing 4.5: COOL Source for Pong

```
actor class Pong {
  Timer t;

  message INIT() {
    register("Pong",self,0) => Mom;
    t = trigger register_timeout() every 2 sec;
  }

  message register_timeout() {
    register("Pong",self,0) => Mom;
  }

  message register_ack(bool reg, string name, actor a, int key) {
    discard t;
  }

  message pong(int num) {
    ping(num) => *;
  }
}
```

4.3 Internode Messaging

The purpose of the third example is to measure the performance of internode messaging. Here, we use a simple "ping-pong" application, where one actor sends a message to another actor, which then responds by sending a message back to the original actor. The program performs this sequence a number of times, each time recording the time difference between sending a message and being activated with the response (round-trip). The COOL code for the "pong" side of this application is presented in Listing 4.5; the COOL code for the "ping" side of this application is presented in Listig 4.6.

To get an appreciation for how well Steam's internode messaging performs, seven different scenarios are executed.

- Scenario 1: Both *Ping* and *Pong* actors reside on the same Steam node. All messages are local. The application runs on machine *S1*.
- Scenario 2: Actor *Ping* resides on node A. Actor *Pong* resides on node B. Both node A and node B are hosted on machine *S1*.

Listing 4.6: COOL Source for Ping

```
actor class Ping {
  Timer t;
  int count = 0;
  int max = 0;
  Pong p;

  message INIT(int maxIterations) {
    max = maxIterations;
    lookup("Pong",0) => Mom;
    t = trigger lookup_timeout() every 2 sec;
  }

  message lookup_timeout() {
    lookup("Pong",0) => Mom;
  }

  message lookup_ack(string name, actor p, int key) {
    discard t;
    p = (Pong)p;
    pong(count) => p;
    start_stopwatch();
  }

  message ping(int seq) {
    if (seq == count) {
      stop_stopwatch();
      count = count + 1;
      if (count < max) {
        pong(count) => p;
        start_stopwatch();
      } else {
        done() => owner;
        terminate;
      }
    }
  }
}
```

Table 4.4: Ping-Pong Round-Trip Performance Summary

	Scenario 1	**Scenario**	**Scenario 3**	**Scenario 4**	**Scenario 5**	**Scenario 6**	**Scenario 7**
Round-Trip Time in μsec to Send/Receive 100 Messages (stdev)	6.73 (0.57)	200101.4 (1007.36)	43.89 (2.38)	153.11 (5.63)	28.54 (10.56)	46.02 (9.82)	145.13 (20.26)

- Scenario 3: Actor *Ping* resides on node A. Actor *Pong* resides on node B. Both node A and node B are hosted on machine *D1*.
- Scenario 4: Actor *Ping* resides on node A. Actor *Pong* resides on node B. Node A is hosted on machine *S1*, and node B is hosted on machine *D1*.
- Scenario 5: *Ping* and *Pong* are coded up as two separate processes using C and UDP instead of COOL/Steam. Both processes *Ping* and *Pong* run on machine *S1*.
- Scenario 6: *Ping* and *Pong* are coded up as two separate processes using C and UDP instead of COOL/Steam. Both processes *Ping* and *Pong* run on machine *D1*.
- Scenario 7: *Ping* and *Pong* are coded up as two separate processes using C and UDP instead of COOL/Steam. Process *Ping* runs on machine *S1* and process *Pong* runs on machine *D1*.

The results of running the seven scenarios are presented in Table 4.4.

4.3.1 Observations

A number of observations can be made based on these results. First, running multiple Steam nodes on a single processor machine yields terrible performance. The reason for this is that the Steam kernel never blocks. When the node (process) running *Ping* sends the message to the node running *Pong*, the node running *Ping* keeps running until the Linux kernel schedules the node running *Pong*. Likewise, once *Pong* is scheduled to execute, it receives the message from *Ping*, activates actor *Pong*, sends a message to *Ping*, and then

keeps running until the Linux kernel schedules the node running *Ping*. Based on the times recorded for Scenario 2, Linux seems to run a task for a maximum time slice of 100 msec. If the Steam dispatch loop were to yield whenever it had no work to do (as mentioned in Section 3.6.5), then Scenario 2 would perform more like Scenario 3. In Scenario 3, both Steam nodes execute on the same machine, but the Linux kernel can execute the two nodes in parallel, one on each of its two processors.

A second observation has to do with the relative performance of both intranode and internode messages. Local messages are an order of magnitude faster than remote messages sent between different nodes that reside on the same (multi-processor) machine. Likewise, messages sent between nodes residing on the same (multi-processor) machine are roughly an order of magnitude faster than messages that actually have to traverse the network.

A final observation is that Steam messaging is comparable to sending "raw" messages over UDP. Scenarios 5, 6, and 7, represent the upper limits of performance possible without the overhead introduced by a system such as Steam. COOL and Steam introduce some overhead, but it appears marginal. In one case however, internode messaging in Steam appears to out-perform "raw" UDP messaging. Sending of 100 messages under scenario 3 appears faster than its counterpart in scenario 6. A possible explanation for this discrepancy is that Steam uses non-blocking sockets, and scenarios 5, 6, and 7 use blocking sockets.

4.4 Timer Accuracy

The purpose of the fourth example is to measure the accuracy of timers. We start by looking at the performance of one-shot timers. Later, we adapt our technique to analyze the performance of periodic timers.

4.4.1 One-shot Timers

The COOL source to measure performance of one-shot timers is outlined in Listing 4.7. To measure the accuracy of a timer, a *OneshotTimerMonitor* actor is allocated. The role of the *OneshotTimerMonitor* is to create a one-shot timer with a random duration (between 10

Listing 4.7: COOL Source for Measuring Accuracy of One-shot Timers

```
actor class OneshotTimerMonitor {
  Timer t;
  int max;
  int iter;
  int r;
  message INIT(int maxIterations) {
    max = maxIterations;
    iter = 0;
    r = random();
    t = trigger timeout() on r msec;
    start_stopwatch();
    commit;
  }

  message timeout() {
    stop_stopwatch(r);
    iter = iter + 1;
    if (iter < max) {
      r = random();
      t = trigger timeout() on r msec;
      start_stopwatch();
      commit;
    } else {
       done() => owner;
       terminate;
    }
  }
}
```

and 15000 msec) and then wait for the timeout message to get dispatched. By comparing the expected duration with the actual time it took for the timeout message to be dispatched, we can quantify its accuracy. Rather than having the *OneshotTimerMonitor* terminate after each timeout message, the *OneshotTimerMonitor* can be configured to repeat this sequence for a given number of iterations.

Table 4.5 summarizes the results of measuring the accuracy of one-shot timers. All tests were run on single-processor machine *S1*. Each run was repeated 5 times.

4.4.2 Observations

The first observation that can be made is that one-shot timers seem to perform quite well. The smallest timer wheel in the timer subsystem operates at a resolution of 10 msec. For

Table 4.5: One-shot Timer Performance Summary

Run Number	Number of Actors	Number of Iterations	Average Difference between expected and actual timeout in μsec (stdev)	Maximum Difference in μsec	Minimum Difference in μsec
1	10	1	1683.7 (28.3)	1736	1633
2	1	10	1069.5 (228.8)	1739	989
3	10	10	1067.5 (209.7)	1733	985
4	100	1	1368.1 (202.2)	1727	1021
5	100	10	1043.7 (268.1)	9189	794
6	1000	1	1447.7 (262.4)	1980	999
7	1000	10	1053.6 (386.6)	21009	79

timers to perform well, all timeouts should occur within 10 msec of their expected duration. From the results, this appears to be the case. For the test cases executed, timeouts generally occur within 10 msec (10000 μsec) of the expected duration, and in many cases within 2 msec.

A second observation is that certain test cases exhibit larger than average differences between expected and actual duration. The test cases that exhibit this characteristic are where the *OneshotTimerMonitors* execute for only a single iteration before terminating. A possible explanation for this behavior has to do with the nature of the actor invocation. If a timer is created as a response to a non-timeout message, it takes longer for the timer to be created than if the timer is created as a response to a timeout message. The first time the *OneshotTimerMonitor* is activated, it is due to a message sent from its owner, and not due to a timeout. Due to the nature of the dispatch loop depicted in Listing 3.2, it could take a few iterations of the dispatch loop before the timer subsystem is called to service a tick. Alternatively, it could take a while to service inbound or outbound messages, before the timer subsystem is called to service a tick. New timers are not actually created (inserted) until the final phase of tick processing, as depicted in Listing 3.1. If a timer is created in response to a timeout, the Steam kernel is already in the timer subsystem, and it will not take as long for the timer to be created. Tests that run for multiple iterations are dominated by timers that are created in response to timeout messages.

A final observation is that the maximum difference between expected timeout duration and actual timeout duration can be larger than 10 msec. Another artifact of running such tests on a system like Linux is that the measured accuracy of timers is subject to scheduling of processes by the Linux kernel. If the process is scheduled out just as a timeout message is about the be dispatched to an actor, it could take tens of msec before the process is scheduled to run again. If a process' maximum time slice is 100 msec, as suggested in the observations of Section 4.3.1, any single timer could be out by as much as 100 msec. This situation is more likely to occur in program runs for a long duration (several seconds). This is likely what occurred where maximum differences of 9189 and 21009 listed.

4.4.3 Periodic Timers

We now turn our attention to periodic timers. The COOL source to measure performance of periodic timers is outlined in Listing 4.8. The structure of this program is similar to its one-shot counterpart. The main difference is that from the COOL perspective, a timer only needs to be created once.

Table 4.6 summarizes the results of *PeriodicTimerMonitor* actors, each set to run for 10 iterations. Running this test case with iterations set to 1 does not really make sense for periodic timers, however, these cases are included so that periodic timers can be compared to one-shot timers. Like the previous test, this test was run on the single-processor machine, *S1*. Each run was repeated 5 times.

4.4.4 Observations

The first observation is that, like one-shot timers, periodic timers appear to perform quite well. Most timeouts are within 10 msec of the expected duration. However, as mentioned in Section 3.3.7, periodic timers are subject to drift. This drift is illustrated in Table 4.7. Here the times for a sample periodic timer are listed. Cumulative drift is about 10 times the difference between expected and actual timeout.

Listing 4.8: COOL Source for Measuring Accuracy of Periodic Timers

```
actor class PeriodicTimerMonitor {
  Timer t;
  int max;
  int iter;
  int r;

  message INIT(int maxIterations) {
    max = maxIterations_;
    iter = 0;
    r = ext_random();
    start_stopwatch();
    t = trigger timeout() every r msec;
    commit;
  }

  message timeout() {
    stop_stopwatch(r);
    iter = iter + 1;
    if (iter < max) {
      start_stopwatch();
      commit;
    } else {
      discard t;
      done() => owner;
      terminate;
    }
  }
}
```

Table 4.6: Periodic Timer Performance Summary

Run Number	**Number of Actors**	**Number of Iterations**	**Average Difference between expected and actual timeout in μsec (stdev)**	**Maximum Difference in μsec**	**Minimum Difference in μsec**
1	10	1	1669.6 (36.8)	1733	1607
2	1	10	1073 (227.8)	1740	994
3	10	10	1068.6 (209.0)	1729	993
4	100	10	1044.7 (424.4)	19035	963
5	1000	10	1053.5 (380.7)	21428	-11

Table 4.7: Sample Drift of Periodic Timer

Time of Timer Creation in μsec since timer created	Time of Timeout in μsec since timer created	Expected Duration of Timer in μsec	Actual Duration of Timer in μsec	Difference in μsec	Cumulated Drift in μsec
0	941490	940000	941490	1490	1490
941490	1882490	940000	941000	1000	2490
1882490	2823480	940000	940990	990	3480
2823480	3764490	940000	941010	1010	4490
3764490	4705480	940000	940990	990	5480
4705480	5646490	940000	941010	1010	6490
5646490	6587480	940000	940990	990	7480
6587480	7528480	940000	941000	1000	8480
7528480	8469490	940000	941010	1010	9490
8469490	9410480	940000	940990	990	10480

A second observation is that, like one-shot timers, periodic timers are sensitive to scheduling decisions made by the Linux kernel. In two of the test runs, the maximum difference between actual and expected timeout is about 20 msec. The impact of this is greater for periodic timers than one-shot timers, as it exacerbates drift.

A final note about the results listed in Table 4.7 is that it appears there may be a minor problem in the handling of timers. In general, the difference between actual and expected timeout should never be less than 0. Out of almost 25000 timeouts that were analyzed, only one had a negative difference. For this one misbehaving timer, the expected timer duration was close to 15 seconds. It is possible that a rounding error occurred during the time in which its insertion locations were pre-calculated. In general, however, having a timer expire 11 μsec early for a timer set to expire with duration of 15 seconds does not seem too serious.

4.5 Simulated Call Control

The purpose of the final example is to demonstrate a distributed COOL/Steam application that combines all of the aspects of Steam demonstrated so far. The subject of this final example is call control. Specifically, this example simulates a subset of the interactions

that take place as part of a Session Initiation Protocol (SIP, RFC 3261) dialog. SIP is a request-response protocol for setting up and tearing down sessions, such as a Voice over IP phone call. This example simulates two scenarios: call setup and call tear-down. Call setup is illustrated in Figure 4.2. Call tear-down is illustrated in Figure 4.3.

Call setup consists of the *caller* sending an *INVITE* message to a *proxy*, which forwards the request to the *callee*, which handles the call. The *callee* responds with an *OK* message. After receiving the *OK* message, the *caller* sends an *ACK* to complete the three message handshake. At this point, the session is considered connected.

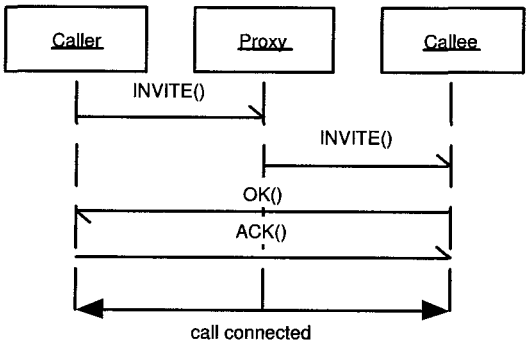

Figure 4.2: Call Setup with SIP

Call tear-down consists of either party (*caller* or *callee*) sending a *BYE* message to its counter part. After receiving a *BYE* message, an *OK* is sent to acknowledge the *BYE*. When a party receives a *BYE* and has sent the *OK*, it can tear-down its end of the connection. Likewise, once a party receives the acknowledgement to its *BYE*, the near end of the connection can be torn down.

Using COOL, each agent (*caller*, *callee*, and *proxy*) can be modeled. The COOL source for the *Caller* is listed in Listing 4.9 and Listing 4.10. The *Caller* places a call by sending an *invite* to the Proxy. A timer is created, which forces the call to be disconnected if any message is lost. Once the *ok* response is received, an *ack* is sent to the *Callee*. At this point the call is considered connected. Once connected, a second timer is created. If this second timer expires before a *bye* is received, the *Caller* initiates the *bye*, waits for the

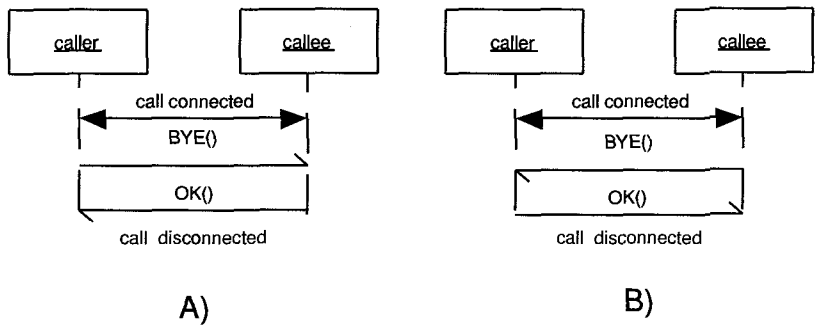

Figure 4.3: Call Tear-down with SIP

bye_ack, and then terminates. If at any time a message is received that is not expected, an error is logged and the invocation aborts.

The COOL source for the *Proxy* is listed in Listing 4.11. The *Proxy* simply receives *invite* messages, creates a new *Callee* actor to handle the *invite*, and then forwards the *invite* to the *Callee*.

The COOL source for the *Callee* is listed in Listing 4.12 and Listing 4.13. Once initialized, the *Callee* expects to receive an *invite* from the *Caller*, to which the *Callee* responds with an *ok*. Once the *Callee* receives the *ok_ack*, the call is considered connected. Like the *Caller*, the *Callee*, creates one timer for handling lost messages, which upon its expiration, disconnects and terminates. Once a call is connected, the *Callee* also creates a second timer. If this second timer expires before the *Callee* receives a *bye* message, the *Callee* initiates tear-down, by sending a *bye*, waiting for a *bye_ack*, and then terminating.

Based upon the above actors, a simulation of call setup and tear-down is performed, where performance characteristics of the following aspects are measured:

- Time for a *Caller* to send an *invite* and *receive* an ok.
- For both *Caller* and *Callee*, the difference between timer duration and how long it took for a timeout message to be dispatched.
- Time for a *Caller* to send a *bye* and receive a *bye_ack*.

Listing 4.9: COOL Source for SIP Caller Part 1

```
actor class Caller [atomic] {
  Proxy proxy = none;
  Callee callee = none;
  Timer t1, t2;
  int callId = 0;

  message INIT(actor p) {
    proxy = (Proxy) p;
    placeCall() => self;
  }

  message placeCall() {
    callId = generateCallId();
    invite(callId) => proxy;
    t1 = trigger disconnect() on 30 sec;
  }

  message disconnect() {
    error();
    terminate;
  }

  message ok(int id) {
    int r;
    if (id == callId) {
      callee = (Callee) *;
      ok_ack(callId) => callee;
      r = random();
      t2 = trigger endCall() on r msec;
    } else {
      error();
      abort;
    }
  }
  ...
}
```

Listing 4.10: COOL Source for SIP Caller Part 2

```
actor class Caller  [atomic] {
  ...

  message endCall() {
    bye(callId) => callee;
  }

  message bye_ack(int id) {
    if (id == callId) {
      discard t1;
      terminate;
    } else {
      error();
      abort;
    }
  }

  message bye(int id) {
    if (id == callId) {
      bye_ack(id) => callee;
      discard t1;
      discard t2;
      terminate;
    } else {
      error();
      abort;
    }
  }
}
```

Listing 4.11: COOL Source for SIP Proxy

```
actor class Proxy [atomic] {
  int numInvites = 0;
  int numCompleteCalls = 0;

  message invite(int id) {
    count = count + 1;
    Callee c = new Callee;
    if (c != none) {
      INIT() => c;
      => c;
    } else {
      error();
      abort;
    }
  }

  message done() {
    numCompleteCalls = numCompleteCalls + 1;
  }
}
```

- Time for *Callee* to send *ok* and receive an *ok_ack*.
- Time for a *Callee* to send a *bye* and receive a *bye_ack*.

To run the simulation, four Steam nodes were required. One node hosted the *Proxy*, and all of the *Callees*. The other three nodes only executed *Callers*. Each node with *Callers* had 100 *Callers*, each attempting to place 10 calls in sequence. At any one time, there could be as many as 300 concurrent calls in progress. Machine *D1* hosted the *Proxy/Callee* node and one of the *Caller* nodes. Machine *D2* and *S1* each hosted one of the *Caller* nodes. A summary of results in presented in Table 4.8 and Table 4.9.

4.5.1 Observations

Prior to this example, all tests were designed to measure specific aspects of Steam in a highly contrived situation. The SIP example attempts to provide numbers based upon a more realistic environment. The numbers presented seem reasonable. For example, it makes sense that the *invite-ok* dialog takes the longest amount of time, since the *Proxy*

Listing 4.12: COOL Source for SIP Callee Part 1

```
actor class Callee [atomic] {
  Caller caller = none;
  Timer t1, t2;
  int callId = 0;

  message INIT(){
    t1 = trigger disconnect() on 30 sec;
  }

  message invite(int id) {
    callId = id;
    caller = (Caller)*;
    ok(id) => caller;
  }

  message disconnect() {
    error();
    terminate;
  }

  message ok_ack(int id) {
    int r;
    if (id == callId) {
      r = random();
      t2 = trigger endcall() on r msec;
    } else {
      error();
      abort;
    }
  }

  ...
}
```

Table 4.8: Execution Summary of SIP Simulation

	Runtime in seconds	**Number of Local Messages**	**Number of Inbound Messages**	**Number of Outbound Messages**	**Number of Timeouts**
Caller* running on *D2	91	1406	2001	3011	528
Caller* running on *S1	107	1406	1998	3004	470
Caller* running on *D1	83	1406	2002	3002	508
Proxy/Callees* running on *D1	120	9019	8999	6002	1512

Listing 4.13: COOL Source for SIP Callee Part 2

```
actor class Callee [atomic] {
  ...

  message endcall() {
    bye(callId) => caller;
  }

  message bye_ack(int id) {
    if (id == callId) {
      discard t1;
      done() => owner;
      terminate;
    } else {
      error();
      abort;
    }
  }

  message bye(int id) {
    if (id == callId) {
      bye_ack(id) => caller;
      discard t1;
      discard t2;
      done() => owner;
      terminate;
    } else {
      error();
      abort;
    }
  }
}
```

Table 4.9: Performance Summary of SIP Simulation

	Average Time in μsec (stdev)	**Minimum in μsec**	**Maximum in μsec**
Caller*: *invite-ok	186.0 (210.0)	53	6380
***Caller* timer**	1627.3 (212.5)	983	2880
Caller* *bye-bye_ack	140.8 (64.2)	46	565
Callee* *ok-ok_ack	166.9 (199.2)	46	6308
***Callee* timer**	1527.0 (300.7)	977	3240
Callee* *bye-bye_ack	152.9 (67.2)	46	386

must first create the *Callee* and forward the *invite* to this newly created *Callee*, before the *Callee* can respond. Likewise, the two types of *bye-bye_ack* dialogs have durations that are comparable. The *bye-bye_ack* dialog for the Callee may take slightly longer because it runs on *D1*, which is the node that must handle the most messages. For the most part, timers performed very well, with all timeouts occurring within 10 msec of their expected duration.

During execution, a small number of errors did occur. These errors related to messages that seemed to be delivered out of order and messages that could not be delivered at all. Out of order messages were detected by the transport subsystem, and subsequently dropped. Messages that could not be delivered occur when a message arrives for an actor that no longer exists. For example, if the *Caller* and *Callee* send *bye* requests at approximately the same time, it is possible that a *Caller* or *Callee* can terminate before receiving a *bye-ack*.

Based on this example, we have demonstrated how location transparency can enhance scalability. Without modification, the same program executed on 4 nodes (3 machines). Using this same approach and with only minor adjustments to the program, it could just as easily run on 5 nodes or more.

Chapter 5

Conclusion and Future Work

Steam represents a framework for building a distributed concurrent application based upon communicating active objects. Steam is suitable to be used standalone or as a virtual machine to support a language like COOL. Using Steam directly, a developer can implement distributed applications supporting dynamic re-configurability, location transparency, and atomicity. Using COOL in conjunction with Steam gives the developer higher level abstractions for defining actors and for ensuring type safety for messages. The development environment provided by the combination of COOL and Steam has provided a foundation upon which other research can be based ([22],[23],[8]).

The thesis has highlighted two of Steams greatest strengths: simplicity and efficiency. Steam promotes a simple programming model, where all actors run to completion within a single thread of control and all inter-actor communication takes the form of asynchronous messaging. Messages are always delivered in order, and apart from overhead introduced by the underlying transport, all inter-actor communication is the same regardless of where the actors reside.

Through experimentation, we have demonstrated that systems based on active objects can be built using Steam. Furthermore, we have shown that applications based upon Steam perform well and are scalable. By minimizing copying of memory and adhering to a simple first-come-first-serve dispatching algorithm, active objects in Steam can outperform similar systems based upon multiple threads of control. When combined with COOL, Steam

only adds marginal overhead to using a raw internode transport. In addition to providing an efficient actor representation and inter-actor communication facility, Steam provides a precise and flexible timer abstraction.

By running within a single thread of control, the programming model for Steam has substituted the need for sophisticated synchronization mechanisms, scheduling, and priorities for simple run to completion semantics where actors are dispatched as fast as possible according to message load. A Steam program has no need for mutexes, semaphores, etc... and due to its asynchronous nature, is exempt from some of the pitfalls of multithreaded programs, such as deadlock. Where applications require priorities for actors or more realtime-like scheduling, Steam offers a few alternatives. Through customizing Steam's configuration parameters, the programmer can specify a priority for dispatching timeout messages over regular actor-actor messages. Likewise, by using periodic timers, a programmer can specify different actors to be scheduled at different set frequencies (subject to slight drift or error, of course), achieving scheduling that approximates that which you would expect of a rate-monotonic scheduler.

Steam is not without weaknesses. There are a number of areas where additional research and development effort could be expended to improve Steam. These areas are listed below:

- Improved accuracy of periodic timers. The problem with periodic timers as implemented is that they are subject to drift, since they are essentially one-shot timers that get re-created upon expiration. An improved implementation that removes this drift would make periodic timers truly periodic. However, there would still be an issue with accuracy of periodic timers because Steam allows for only one thread of control. If an actor is executing when a timer should have expired, the timeout message from the timer could not be dispatched until the actor currently executing completes its invocation. Improvements could be made where if the timer subsystem knows that a timer is just about to expire, it could busy wait until that timer expires, preventing any actor from being invoked until after the timeout has been handled.

- Multithreading of Steam. Though having a single thread of control is viewed as one of Steam's advantages, introducing multiple threads of control into Steam has been considered. For example, if actors are required to perform blocking operations or if Steam has to interact with other software that does not conform to Steam's asynchronous nature, multithreading could be employed. This could be done by allowing multiple Steam runtime systems to be spawned as separate threads within the same process space. Each runtime would have its own actor subsystem and messaging subsystem, but would be able to share a common timer subsystem and transport subsystem. Different instances of the Steam runtime within the same process could also been assigned different priorities, to reflect the relative importance of actors running on each instance.

- Opening a closed system. Related to the idea of multithreaded Steam is the idea of allowing non-Steam entities to inject messages into Steam. Steam is somewhat of a closed system. There is currently no means by which a user can interact with Steam (via a console) or for Steam to receive interrupts from external events. To accomplish this, a separate thread of control that can send and receive Steam messages could be developed. This separate thread could serve as a proxy, shuttling messages between Steam and a foreign agent.

- New Transports. Steam was originally conceived as a runtime system to support the control of an ATM switch. The size of properties of messages in Steam map quite naturally to ATM cells. It would be interesting to develop alternative transports for Steam (other than UDP). Candidate transports include ATM, Firewire, and SCTP.

- IDL Framework. Steam can be viewed as the virtual machine for COOL. COOL is a great language for prototyping, designing, and experimenting with Steam. An alternative to COOL would be to develop an interface definition language (IDL). Actor definitions and format for messages could be specified in the IDL, and the IDL could generate all the code for handling type safety and marshalling/un-marshalling of data to/from messages.

- Publish and Subscribe Services. One of the criticisms of systems like Steam is that the identities of different agents must be known before communication can take place. For example, for one actor to communicate with another actor, it must first learn the PID of the other actor. Publish and subscribe is a methodology where by identities of such agents need not been exposed. Instead, subscriber agents register an interest in a certain subject with publisher agents. When information that a subscriber is interested in is made available, the publisher sends this information to all subscribers. Implementing a generic publish and subscribe service framework with Steam/COOL, should be quite straightforward.

- Debugging Tools. Developing applications for Steam is quite different from developing traditional applications. Consequently debugging Steam presents new challenges to the programmer. In addition to debugging sequential execution within the context of each actor, the developer has to contend with monitoring protocols implemented via messages sent between actors.

References

[1] John Pugh. Actors–The Stage is Set. *ACM SIGPLAN Notices*, 19(3):61–65, March 1984.

[2] Mantis H.M. Cheng, Gordon W. O'Connell, and Paul Wierenga. COOL : A Concurrent Object Coordination Language. Technical Report DCS-267-IR, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, July 2001.

[3] W. Older. Crisp Kernel Specifications (Release 3). Technical report, Computing Technology Laboratory, Nortel Networks Inc., Kanata, Ontario, Canada, July 1996.

[4] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). In *USENIX Technical Conference*, San Diego, United States, January 1996.

[5] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Pattern Languages of Programming Conference*, Illinois, United States, September 1995.

[6] A. W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software Practice and Experience*, 11(5):435–466.

[7] D. P. Reed J. H. Saltzer and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288.

[8] Gordon W. O'Connell. *A Concurrent Object Coordination Language: Semantics and Applications*. PhD dissertation, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, 2004.

[9] Dennis G. Kafura and R. Greg Lavender. Concurrent Object Oriented Languages and the Inheritance Anomaly. *Parallel Computers: Theory and Practice*, pages 165–198, 1994.

[10] Svend Frolund. *Coordinating Distributed Objects, An Actor-Based Approach to Synchronization*. MIT Press, Cambridge. Massachusetts, United States, 1996.

[11] Michael Papathomas. Concurrency Issues in Object-Oriented Programming Languages. In D. Tsichritzis, editor, *Object-Oriented Development*, pages 207–245. Centre Universitaire dInformatique, University of Geneva, 1989.

[12] Manibrata Mukherji Dennis Kafura and Greg Lavender. ACT++ 2.0: A Class Library for Concurrent Programming in C++ using Actors. *Journal of Object-Oriented Programming*, 6(6):47–62.

[13] Daniel Charles Sturman. Fault Adaptation for Systems in Unpredictable Environments. Master's thesis, Department of Computer Science, Cornell University, Ithaca, New York, United States, 1991.

[14] Miro Samek. *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*. CMP Books, California, United States, 2002.

[15] Carl Hewitt. Viewing Control Structures as Patterns of Message Passing. *Journal of Artificial Intelligence*, 8(3):323–364.

[16] Carl Hewitt and Gul Agha. Actors: a conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.

[17] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge. Massachusetts, United States, 1986.

[18] Gordon W. O'Connell and Mantis H.M. Cheng. A COOL Language Report. Technical Report DCS-268-IR, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, July 2001.

[19] Gordon W. O'Connell. Applications Development in COOL. Technical Report DCS-282-IR, Department of Computer Science, University of Victoria, Victoria, BC, Canada, February 2004.

[20] Gordon W. O'Connell. A Translational Semantics of COOL. Technical Report DCS-281-IR, Department of Computer Science, University of Victoria, Victoria, BC, Canada, January 2004.

[21] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998. North Holland, 1977.

[22] Anthony Justin Howe. Client Migration in a Continuous Data Network. Master's thesis, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, 2002.

[23] Sinesie Calin Somosan. Parallel Constraint Propagation. Master's thesis, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, 2001.

[24] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2000.

[25] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., 1996.