Fall 2007

A light-weight middleware framework for fault-tolerant and secure distributed applications

Ian Jacob Baird

A LIGHT-WEIGHT MIDDLEWARE FRAMEWORK FOR FAULT-TOLERANT AND SECURE DISTRIBUTED APPLICATIONS

by

IAN JACOB BAIRD

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI–ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2007

Approved by

Bruce McMillin, Advisor

Ann Miller

Fikret Ercal

ABSTRACT

This thesis outlines the design, implementation, and performance of a lightweight middleware framework for interprocess communication with an update log propagation algorithm. The system was designed and implemented using a point-to-point based lightweight middleware framework and compared to a similar system implemented utilizing CORBA. The implementation difficulty and performance of two model problems were compared and the efficiency of the lightweight middleware framework was found to exceed that of the traditional CORBA-based solution, while also having the advantage of hiding more of the implementation complexity from the application developer. While the lightweight middleware framework based solutions were more efficient and created smaller message sizes for equivalent message payloads, the CORBA-based solutions performed better with respect to raw message passing performance. Subtleties involved in the underlying network protocol and large amounts of concurrency made the framework very difficult to implement. Issues with concurrency in the interpreter used could possibly hinder the scalability of a solution utilizing the lightweight middleware framework on multi-core hosts.

ACKNOWLEDGMENT

I dedicate this work to my wife Stacy Baird, and to my parents, Barbara and Jason Baird. Without their support, this work would have been impossible.

I would also like to thank Dr. Bruce McMillin, for without his guidance and support I would have never successfully explored the realm of graduate level Computer Science. Any acknowledgement would be incomplete without an expression of my utmost gratitude to my committee members, Dr. Ann Miller and Dr. Fikret Ercal, whose guidance I am incredibly grateful for. I would also like to acknowledge Dr. Mariesa Crow and the NSF IGERT Fellowship, which provided the stipend and funding which made my post-graduate education possible.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The government, business, and scientific communities are becoming increasingly more dependent on middleware, which is defined as a set of distributed system services which have standard programming interfaces and protocols and sit in a layer above the OS and networking software and below applications.[2] The ability to harness the power of software objects using a common Application Programming Interface (API) to mask the complexity of the messaging is both convenient and increasingly necessary for the implementation of large-scale complex systems. PDLS, or the Python Distributed Logging System is an efficient method of interprocess communication and communicating updates to the global state of a distributed system using update log propagation. Events are disseminated throughout a middleware system in response to an external or an internal stimulus. Whereas other systems such as the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA)[11] have similar methods for propagating events throughout a system, it has been shown to be less inefficient and less expedient than PDLS when used in a large-scale environment.

1.1. MESSAGE-PASSING

Computer control structures can be interpreted as "patterns of passing messages"[4] within the context of a distributed system. Message passing systems are also known as "shared nothing systems", in contrast to distributed systems which utilize shared memory or state common to the processors to facilitate inter-process communication.

A message is defined as a collection of data objects, and the structure of a message is defined by collaborating application processes. In a heterogeneous system, data objects within a message are usually typed in order to facilitate conversion of the contained data. Messages may also contain system-dependent control data, such as message length, checksums, or flags, and will always contain a fixed or variable length message body, which holds the system data objects. These messages are composed and passed to a transport service, facilitating delivery of the messages between disparate processes in the distributed system. Generally the transport service

will provide send and receive primitives to alternately send a message (or a set of messages) or to receive a message (or a set of messages). These transport service primitives are well-defined and all of the communicating member processes of the distributed system are contractually bound to abide by the semantics of the transport service primitives. The communications primitives may be direct or indirect, buffered or unbuffered, reliable or unreliable.[3]

1.2. MIDDLEWARE

In the context of a distributed system, middleware is defined as "the software layer that lies between the operating system and the applications on each site of the system."[6] The fields of business and scientific research depend on middleware to facilitate the communication of disparate nodes (also referred to as processes) on a heterogeneous network, often without regard to processor architecture, network connectivity, or network type. Examples of well-known middleware implementations include CORBA by the OMG, the Distributed Computing Environment (DCE), and Distributed Component Object Model (DCOM) by Microsoft, Inc. All of these have gained widespread adoption and use and are well-known by developers and researchers alike.

1.2.1. CORBA. The Common Object Request Broker Architecture (CORBA) is a system of middleware which is defined by a group of specifications published by the Object Management Group (OMG). The goal of CORBA is to provide a standardized framework facilitating the interaction of disparate objects in a location-transparent, hardware, network, and operating system agnostic fashion. It accomplishes this goal by defining the interface via which the objects can communicate using the Interface Description Language (IDL), a series of language-specific mappings for the data-structures and services defined in the IDL. CORBA also uses a higher level abstraction of the message-passing protocols, known as the General Inter-ORB Protocol (GIOP).[17]

The Internet Inter-ORB Protocol is the only mandatory protocol in the CORBA suite. It is in fact defined as GIOP encapsulated by the TCP/IP protocol. However, GIOP can be used with any message-passing protocol as long as the transport meets a well-defined set of specifications[17] In order to be suitable for use by GIOP, it

requires:

- The transport protocol must be connection-oriented.
- Reliable-delivery (byte-order preservation and delivery acknowledgment services must be available) is assured.
- The participants must be notified in cases of connection loss.
- A connection must be initiated using a TCP-like initiation sequence.

GIOP defines a Common Data Representation, known as CDR which encodes all of the datatypes defined in OMG IDL. The specification is endian-safe and alignment-neutral, allowing messages to be decoded more easily by machines on heterogeneous networks. The combination of IIOP and CDR allows all CORBA Object Request Brokers (ORBs), no matter which vendor implements and supplies the middleware layer, to interoperate and facilitate object communication.

At the heart of any CORBA-based system is the Object Request Broker (ORB). The ORB provides the context in which an object is instantiated, brokers inter-object messages, and resolves objects references at runtime. The ORB also handles inter-ORB communication, using the the GIOP protocol described previously. An ORB also manages the lifecycle of any CORBA object in its context. In the diagram below, the "stub" and "skel" portions of the diagrams represent the automatically generated code, which the CORBA tools create to glue an object or its proxy to the ORB.

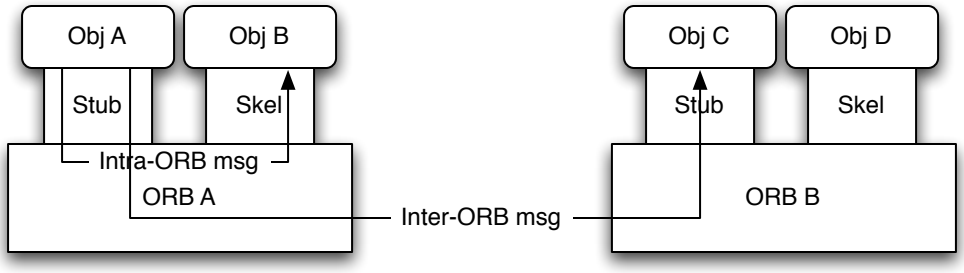

Figure 1.1. Intra-ORB and Inter-ORB Messaging

1.3. MIDDLEWARE AND MESSAGE-PASSING RELATIONSHIP

Middleware utilizes the primitives exposed in the message passing protocols (such as TCP/IP) to encapsulate standardized messages which are then propagated throughout a distributed system. For example, CORBA is able to utilize network protocols such as TCP/IP, UDP, and IPX/SPX to propagate GIOP (General Inter-ORB Protocol) messages between distributed CORBA ORBs.

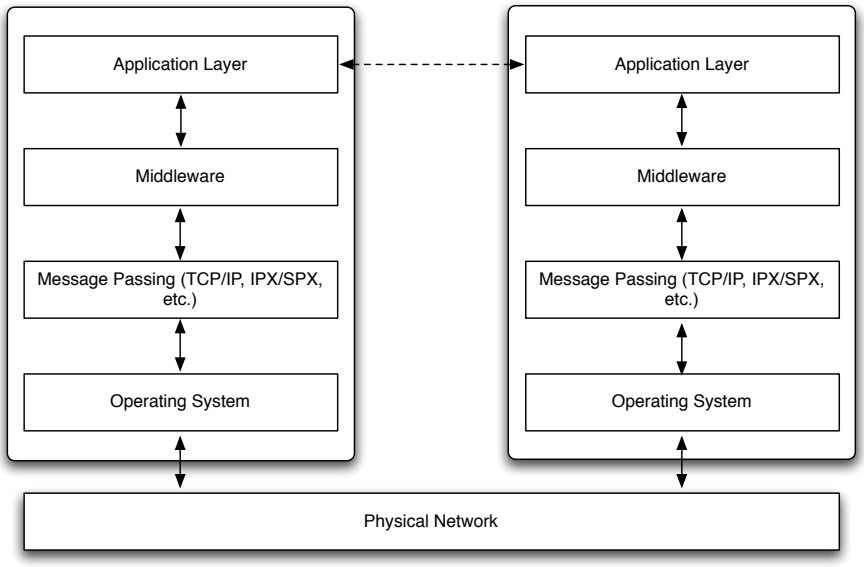

Figure 1.2. Abstract Middleware and Message-passing on Two Distributed System Nodes

1.4. PYTHON

Python is an interpreted, object-oriented language and runtime. It was inspired by the language ABC, a teaching language created in the early 1980s aimed at non-professional developers. Python was meant to be a descendant of ABC that would appeal to UNIX/C developers. [7]

The first implementation of Python was created on a Macintosh. However, as the language grew in popularity, it was quickly ported to Linux, Solaris, FreeBSD, Microsoft Windows, and eventually to Mac OS X. One of the motivations was to make Python follow UNIX conventions and rely on the UNIX infrastructure, without tightly binding the runtime to the UNIX platform.

One of the most controversial features of Python is its use of indentation to denote functional blocks of code and scoping. Most languages use delimiters such at parentheses and brackets to delimit scoping blocks, while Python's interpreter garners all of this information from the indentation of the line and following lines of code. This is justified by the supposed readability benefit gained by the language when whitespace is not only encouraged, but strictly enforced by the interpreter and runtime. The design decision is justified by the theory that object reuse is more easily facilitated by readable and easily understandable computer code, which is encouraged by the use of whitespace in the Python language.

One of the other features of Python, which non-professional and beginning developers enjoy, the correct way of coding a logical construct is generally straightforward. Unlike Perl and other language that pride themselves on having multiple ways of accomplishing equivalent logic, Python strives to have a single, obvious way to correctly accomplish the task. This reduction in variability is alleged to increase the familiarity of developers with a construct, making code easier to understand. This may also lead to benefits in code maintenance, as most applications in research or in industry have a lifetime that extends beyond the involvement of the original developers.

As an interpreted language, Python trades execution speed for an increase in programmer efficiency.[14] Over the course of the BOOTS[13, 12] system (a model problem to study computer security) implementation and the BOOTS2 system implementation, we found this to be true, as it was possible for one developer to achieve in 3 years what had taken a team of graduate students nearly a decade of combined effort.

1.4.1. Why Python?. Python was chosen as the implementation language for the system we developed due its ease of development and lower cost of maintenance[14]. The other system in use was the CCSP system[8], which was modeled after Hoare's Communicating Sequential Processes (CSP)[5]. CCSP was originally implemented using lex and yacc, which compiled out to an intermediate C representation, which in-turn was compiled into an executable application. The levels of indirection involved in the multiple-stage compilation made the coding of a complex system such as the BOOTS simulation nearly unmaintainable.

1.4.2. Python Pickle Construct. The analogue to CORBA's CDR is the Python "pickle". A pickle in Python is a serialized object graph, which preserves not only the data of the serialized objects, but the relationships between the objects. This is serialized into an endian-safe and alignment-neutral datastream, which is suitable for transmission between disparate nodes on heterogeneous networks. However, Python's pickle construct is specific to the Python runtime, and is unable to interoperate with applications developed in other programming languages or executing other non-Python runtimes.

1.5. PYTHON DISTRIBUTED LOGGING SYSTEM (PDLS)

The Python Distributed Logging System is an implementation of an update propagation system implemented in Python and C. The message-passing primitive used is based on the ADA network rendezvous/select primitives. It was originally implemented to serve the Security Group at the University of Missouri - Rolla. PDLS attempts to retain the simplicity of Python, while providing the power of middleware. This design decision was made to ensure that the application developer is not overburdened by the orthagonal concerns of state update, auxiliary communication, and log maintenance.

2. SURVEY

The following is a brief survey of middleware and message-passing systems, with emphasis given to those more suited to fault-tolerant and secure distributed systems.

2.1. WEAKLY CONSISTENT MESSAGE PASSING TECHNIQUES

A message-passing system is considered to be "weakly-consistent" if the consistency constraints on the replicated data are loose. Sites throughout the distributed system may update or see the replicated data. The system need not support serializablity and the "most-recent update" of the data is considered to be good enough. One example of a system with requirements fitting a weakly-consistent system is the Lotus Notes system, which is produced by IBM. In this system there is the requirement to propagate updates (in the form of postings) throughout the system. The only constraint on these postings is that of causality, making it possible for the end-users to follow the thread of conversations.[3]

2.1.1. Epidemic Techniques.

When dealing with communications in a distributed system, one cannot assume the availability of a reliable communication channels and accept the idea of limited communication. In such a system, one must be able to quickly and effectively spread the news of an update without overwhelming the available communications infrastructure with messages. Thus, the epidemic algorithm is used to accomplish this goal.

Given a data item *d*, a node may update the data, creating update *u(d)*. As soon as a node in the distributed system learns of an update, it should immediately begin to attempt to propagate the update to the rest of the system. If the node learns that *u(d)* is well-known during the course of propagating *u(d)*, the node should attempt to update the other nodes less vigorously than before. Given *u(d)*, servers are categorized as:

- Infectious - The server has knowledge of *u(d)* and is actively propagating it.
- Susceptible - The server does not know of *u(d)*.

- Removed - The server has knowledge of *u(d)* and is no longer vigorously propagating it.

Given the previous categorizations, the algorithm follows naturally:

- 1. Infected server learns of *u(d)* and is categorized as infectious.
- 2. The infectious server contacts random servers and attempts to propagate *u(d)*.
- 3. If an infectious server contacts another infectious or removed server, it has a 1/k probability of being removed.

The algorithm progresses through multiple cycles, with the goal of the susceptibility of each node rapidly converging to zero. At this point, *u(d)* is considered to have propagated throughout the system. The epidemic algorithm is well-suited for initial distribution of an update, but fails when attempting to infect the remaining few susceptible nodes. Due to this well-known limitation of the algorithm, a backup algorithm is often used.[3]

2.1.2. Anti-Entropy.

Anti-entropy is a simple form of an epidemic algorithm. The effect of the anti-entropy algorithm can be either push, pull, or push-pull, depending on its design.[1]

- Push - propagates all new updates to the remote node from the local node, replacing remove value for update with timestamp less than those node to the local node.

- Pull - retrieves all new updates from the remote node, replacing local values for updates with timestamps superseding those known to the local node.

- Push-pull - combines the two operations above, involving both a local and remote comparison of updates. Updates on the local and remote nodes are bidirectionally compared, and those older updates on either node are superseded by the newer ones using timestamp comparison.

Due to properties of the anti-entropy algorithm described in the Xerox PARC report, pull or push-pull is preferable in conditions where few susceptible nodes remain, as a node's susceptibility converges more rapidly to zero in the case of pull-push and pull than in the purely push case.[1]

Anti-entropy can be a very expensive technique, especially in the case of the push-pull algorithm, as two entire database comparisons are required to perform the push-pull update.

2.1.3. Update Log.

The epidemic algorithms outlined earlier assume that when an update *u(d)* of data item *d* is propagated to a node, that the value carried or implied by *u(d)* completely overwrites the value of *d*. However, this is not optimal in all cases, as some data items are more correctly understood as a series of causally ordered updates (a history) and an initial value. In this scenario, in order to properly understand the value of *u(d)*, it is imperative that the entire history is correctly propagated and applied to the initial value of the data item. Epidemic algorithms they give only probabilistic guarantees of the propagations of updates, and are therefore incompatible the constraints governing a system based on the dissemination and application of correct histories.

The update log propagation algorithm relies on each node keeping a log L of all of the updates it has seen. L is composed of a partially ordered list of events. Each event e is comprised of the following fields, which corresponds to an update:

structure member	description
e.method	method and causally associated parameters
e.VTS	associated vector timestamp
e.pid	id of processor which executed the operation

Table 2.1. Casual Log Event Record Composition

The log exchange occurs between two communicating processors and implies the constraint that the exchanged logs must be consistent. A log is considered to be consistent if for an event e executed on processor p every processor $j = 1 \ldots \mathrm{M}$ and every event f:

$$f \in L_p[e] \leftrightarrow f \in Lj[e]$$

An event's context is represented by a vector timestamp, containing the perceived values of clocks of the rest of the nodes in the system at the time of the update. When exchanging logs, all of the events causally preceeding the current event are passed along in the log.

In the interest of efficiency, matrix timestamps are used to garbage-collect event logs. The matrix timestamp is maintained at each node and contains the perceived values of the vector timestamps of the rest of the nodes. This is useful for two reasons. First, if a node knows that an event has been propagated to all of the other nodes in the system, the event can be safely removed from the log. Secondly, when propagating the update log, the node must only send the events the recipient has not yet learned of, creating another efficiency in the log propagation algorithm. Each row in the matrix timestamp is an expression of the lower-bound of the event log in a remote node or the local node. After each log propagation and exchange, this lower-bound is updated, as the vector timestamps and node ids attached to the events received will tell the integrating node the state of the remote node's event log.

The log distribution protocol does not specify a method for determining which processors will propagate their logs. It is perfectly acceptable to use anti-entropy or

another epidemic algorithm to determine this. Also, distribution of the log and the lower-bound vector timestamp are able to be separated. Matrix timestamp distribution can be expensive, as it is always M^2 in size.

PDLS utilizes the update log propagation model, where the operations are the serialized remote procedure calls and log exchanged is performed during the execution of the network rendezvous or select primitive.

2.2. FTCORBA

Fault-Tolerant CORBA (FTCORBA) describes a set of services, an architecture, and a set of mechanisms which are composed to form a framework for highly-available, resilient, distributed systems. The applicability of FTCORBA runs from large scale medical systems to small real-time embedded systems used in monitoring systems and medical equipment. FTCORBA is invasive, in other words applications must actively cooperate with the framework and be aware of its presence in order to reap the benefits of FTCORBA.

FTCORBA encompasses three main features: *entity redundancy*, *fault detection*, and *fault recovery*. However in this work, we will mainly concentrate on the mechanics of the entity redundancy service, as this is where the services pertaining to object and data replication are located.

Entity Redundancy is achieved via the replication of CORBA objects. An object group is utilized to replicate the CORBA object, with each object within the group implementing a common interface. In this respect, clients are unaware of the replicated nature of the endpoint and use the replicated object as if it were a standard CORBA object. FTCORBA designates the object group with an Interoperable Object Group Reference (IOGR). The IOGR is maintained by FTCORBA throughout the lifecycle of the replicated object and is used by the client to call services on the replicated object.

FTCORBA defines three styles of replication: *stateless*, *active*, and *passive*. These styles are differentiated by the points where an object groups' member objects reach a consistent state and the mechanisms used to create the consistent state.

Stateless replication styles carry all of the information required to complete the invocation or a pointer to an external location where the pertinent state can be

retrieved with the invocation. This style of replication is suited to objects that do not persist state between invocations.

Active and passive replication are used when objects need to maintain some state between invocations. In the case of passive replication, an object within the object group is designated as the primary object and its state is periodically queried and logged out to persistent storage. If and when a failure of the primary object is detected, a backup object is promoted to primary status within the object group and is brought up to date by reading the persisted log of the previous primary object.

In active replication, each object within the group processes the request, therefore maintaining a consistent state across all members of the object group. The client receives only one response from the object group because the ORB filters duplicate responses from the group.

One of the main weaknesses of the FTCORBA approach is its reliance on reliable communication between all members of the object group in order to allow for coordination and state synchronization. FTCORBA does not address faults related to network partitioning (unreachable nodes in the network), commission faults (incorrect results from the execution of an invocation using a faulty or compromised object), and correlated faults (i.e. application development logic errors).[9]

3. TECHNIQUE

In order to test the performance of a Python-based update-log propagation solution against the overhead and constraints of a CORBA-based solution, two systems were constructed and applied to the model problems.

3.1. PYTHON DISTRIBUTED LOGGING SYSTEM

The Python-based update log propagation solution, known henceforth as the Python Distributed Logging System (PDLS), grew out of the CCSP system [8]. It utilizes the update log propagation algorithm to distribute global state and implements a subset of Ada-style network communication and synchronization primitives.

3.1.1. The Architecture of PDLS. PDLS is architected in a layered fashion, where ideally each layer need not know of the layers above it and only depends on the layers below it to operate. This design methodology proved to be useful for testing, as each layer could be tested in independently from the layers that depended on it, starting at the core of the application (the network layer), and moving up to the top layer (the event layer).

PDLS has 3 main layers (referred to here as "services"). They are as follows:

- NetworkService - Handles all of the network I/O of the application. Utilizes BSD sockets for all network I/O tasks. All I/O in this layer is asynchronous and is implemented using a thread-pool of worker threads which consumes a queue of work items. When a work item is completed, the thread-pool notifies the event subscribers of the completed I/O.

- RendezvousService - Provides support for the rendezvous and select primitives (styled after Ada). Contains a dictionary of tags (which is a variable name plus the label of the expected node). The RendezvousService receives notifications from the NetworkService and utilizes it exclusively for all network I/O.

- EventService - Utilizes the RendezvousService to provide event logging services to the application. The application notifies the EventService of any internal updates and uses EventService provided primitives to wrap inter-node RPCs. The EventService satisfies the RPC and handles the propagation and maintenance of the event log.

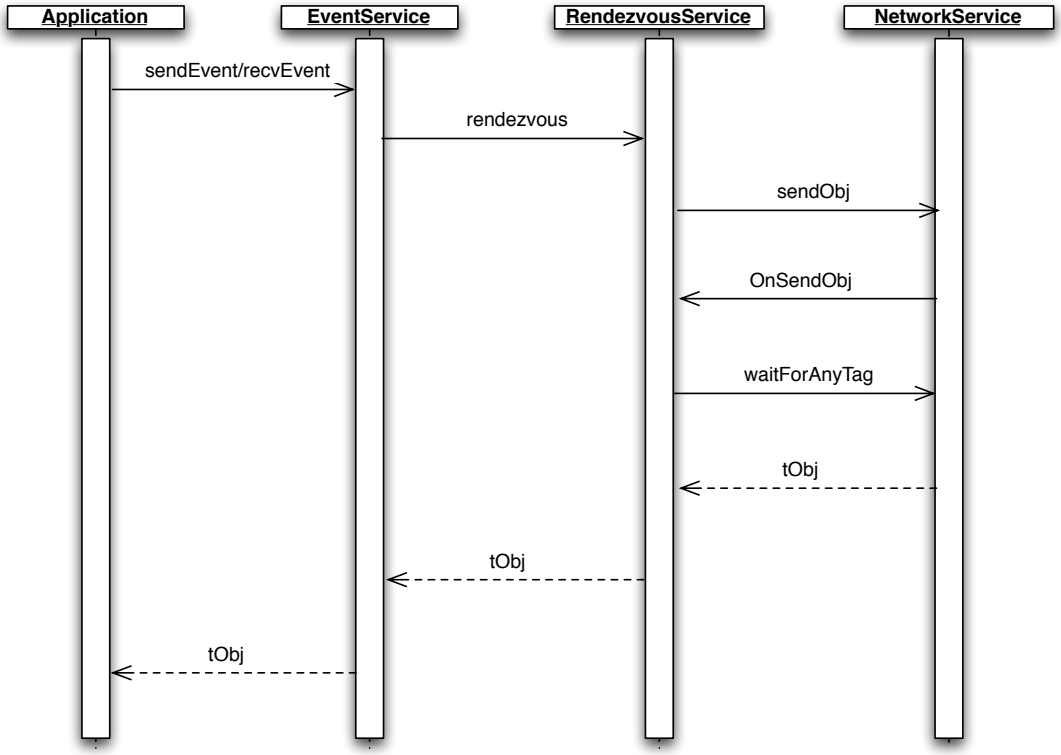

Figure 3.1. Sequence Diagram Showing a Typical Interaction between the Application and the PDLS Service Layers

3.1.2. Message Passing. PDLS packages all of its messages in packets, known as PicklePackets within the system. A PicklePacket utilizes the Python cPickle module to provide all of the data-marshalling support and derives its name from the module. PicklePackets are simple data structures that are constructed as shown in the table below:

Member	Description	Type
seqNum	The sequence number of the packet	integer
keepAliveConnection	NetworkService should keep-alive connection	boolean
payload	A packed-string representation of the packet's payload	string

Table 3.1. PicklePacket Data Structure Composition

The PicklePacket was designed to be a generic carrier of a packed-string representation of a data item update in the system. The inclusion of the keepAliveConnection flag was an implementation detail which was discovered to be necessary due to the overhead associated with building up and tearing down a TCP connection for each message passed in the system.

In the context of PDLS, the generic PicklePacket object generally carries a TaggedObject as its payload. A TaggedObject is a simple structure containing a Python object (type does not matter), a destination globally unique identifier (GUID), and a source GUID, and a data tag. The GUIDs are also known as "process identifiers" or "node labels" and should be considered equivalent within the context of the PDLS system. The data tag is the name associated with the object update which can be interpreted as an instance method, or simply an update to the global state. The members of the TaggedObject structure are enumerated in the table below:

Member	Description	Type
guid	The destination GUID of the tagged object.	string
srdGUID	The source GUID of the tagged object.	string
tag	The data tag.	string
containedObj	The contained Python object.	object instance

Table 3.2. TaggedObject Data Structure Composition

In the current implementation, the data-channel utilized by the message-passing system is TCP/IP and the API used is the BSD sockets API. The NetworkService has been successfully ported to Solaris, Win32, FreeBSD, Linux, and Mac OS X (Darwin). The Network Service could be extended to any arbitrary protocol given that it provided guaranteed delivery of data, as is the case with TCP/IP, and allowed for point-to-point communications between the participating nodes.

3.1.3. Network I/O Implementation. Network I/O in the PDLS system is based upon the NetworkService layer. The NetworkService layer uses a specialized thread-pool implementation which provides cross-platform, operating system agnostic support for asynchronous socket-base network I/O. A thread-pool is used in situations where one desires concurrency and multiple threads, but does not want to incur the performance penalty of constantly setting up and tearing down operating system threads. A Network Service takes a TaggedObject, embeds it in a PicklePacket as the payload, and transmits it using TCP/IP to the peer node. The algorithm for transmitting a packet follows below:

Data: worker thread
Data: thread pool
Data: queue
Result: Assigning Work Items to Worker Threads in the Thread Pool
initialization
while *not shutting down* **do**
    read work item from queue
    if *idle worker thread exists* **then**
        assign work item to worker thread
    else
        spawn new worker thread
        add worker thread to thread pool
        assign work item to worker thread
    end
end

Algorithm 1: Network Thread Pool Work Item Assignment Logic

The Network Service also spawns another thread which is tasked with listening to the prescribed port and responding to remote connections. When a remote connection is accepted, it is passed off to a worker thread, which spins in a select loop, reading all available data until the connection is terminated. If the connection experiences an error, or the last PicklePacket to be consumed from the remote node specifies a value of *false* in the *keepAliveConnection* structure member, the connection is terminated.

Clients of the NetworkService interact by first instantiating the NetworkService and assigning it a TCP port and subscribes to the **SEND**, **RECEIVE**, and **ERROR** events of the NetworkService with custom, client-supplied callback methods. Then the NetworkService enters the listening loop described above. After the NetworkService has been properly initialized, clients utilize the *sendObj* primitive to place work items on the queue and wait for their callback methods to be invoked by the NetworkService. The *sendObj* primitive expects to receive a TaggedObject as a parameter and immediately returns after it has scheduled the send. When a PicklePacket is successfully decoded and the contained TaggedObject is extracted, a **RECEIVE** event is detected by the middleware and all of the subscribers' registered

callback methods are invoked, passing the TaggedObject as the data parameter of the method.

3.1.4. Name Resolution. Currently, the NetworkService is limited to a fixed set of nodes, the cardinality or labeling of which is static during runtime. For this purpose, there is a NameTable utility class which the NetworkService utilizes for all endpoint description queries. The NetworkService passes the Globally Unique Identifier (GUID) of the node it wishes to contact to the NameTable utility, which looks it up in a static map and resolves it to the endpoint description (IP address, and TCP port). This tuple is returned to the caller, or if the GUID is not found in the map, the NameTable utility throws an exception. NameTable merging and update facilities are available, and the intention is for future versions of the system to treat the NameTable as another piece of the global state and utilize the EventService to manage the update and propagation of the updates.

3.1.5. Network Rendezvous and Select. The network rendezvous and select primitives were adapted from Ada and serve as PDLS's primary method of data exchange, as well as transparently performing the propagation and maintenance of update logs (auxiliary communication). For example to perform a simple rendezvous, Node A will create a TaggedObject with the tag "datatag1" and will enter the rendezvous method of the NetworkService. The RendezvousService will check the locally maintained data tag table, find that there is no tag corresponding to "datatag1", and suspend the thread. Node B, running on another processor or locally on a multitasking operating system, will create a similar TaggedObject with the tag "datatag1" and enter the Rendezvous service using the rendezvous method. At this point, the NetworkService will check the data tag table, again locally maintained, and find a tag corresponding to "datatag1". The tags are previously agreed upon and prefixed with the GUID representing the intended recipient node, thus guaranteeing uniqueness throughout the system's runtime. The object tagged with "datatag1" will be immediately returned to the caller and the node will proceed. The object tagged with "datatag1" will be similarly received by Node A, which will wake up, find the tag in the data tag table, and return the object sent by Node B to the caller.

A client node will utilize the select primitive when it might possibly synchronize with more than one remote node. This is useful in situations where there is

one producer, and multiple possible consumers of the data (i.e. a web server). For example, Node A will create a list of TaggedObjects and pass them to the selectObj method of the NetworkService. The NetworkService will then attempt to perform a rendezvous, based on any of the tags passed by the client. However, if it is unable to perform a rendezvous, it will suspend the calling thread and wait for more tags to be inserted into the tag table. There is no order or preference given to any of the possible rendezvous tags. It is a non-deterministic, winner-take-all synchronization. One remote node will be selected from a pool of possible rendezvous partners, and the rest are forced to continue waiting. This has the potential of unfairly favoring a more persistent client, but this was ameliorated by randomizing the order of the list of TaggedObjects passed to the selectObjs method of the NetworkService. Unfortunately, this still leaves PDLS vulnerable to the issue of process starvation in the scheduler.

3.2. EVENT PROPAGATION USING CORBA ORB INTERCEPTORS WITH TAO

CORBA, as the specification currently stands, lacks a facility or a service (i.e. the COSEventService) for the update log propagation, lazy database implementation, or the maintenance of weakly consistent updates to global state. However, in the CORBA 3 standard, there is a "Portable Interceptor" standard, which is suited for the implementation of such a facility. In order to implement a PDLS-equivalent update log propagation facility in CORBA, we implemented a set of client and server request interceptors and tested them against the model problems using the TAO CORBA ORB. While the interceptors provided a way to use "out of band" communications, piggybacking on the IIOP communication stream, there was still the problem of a lack of a suitable event-propagation implementation in C/C++. To address this need, we implemented a C/C++ compatible event log propagation library, known henceforth as "libLazyDB".

3.2.1. libLazyDB Implementation and Design. libLazyDB is a simple C++ library which implements the update log propagation algorithm. It is implemented using the C++ Standard Template Library and is portable to any POSIX-compliant platform. The interceptor library contains utilities for marshalling and

unmarshalling the data to and from event logs into the CORBA CDR representation.

3.2.2. Interceptor Implementation and Design. One of the interesting, and much-touted parts of CORBA is that a client cannot tell by either the object reference or the form of the invocation whether an invocation's target object is local or remote. This is by design, and provides the location-transparency that the designers of CORBA wished to achieve. However, in the case of update propagation, this creates a problem for the implementor of the algorithm, as without non-portable addons to the CORBA specifications, it is impossible to discern during the request phase of a CORBA RPC which endpoint will service the RPC. Therefore, when making a request, the sending node must bundle and send all of the possible update logs (one per node in the system), instead of sending only the pertinent update log.

Input: q such that q is the destination node
Output: LL' such that LL' is a set of propagated logs

initialize LL' to $\emptyset$
for $j = 1 \ldots M$ **do**
  initialize L'_j to $\emptyset$
  foreach $e \in L$ *such that* $min\_TS[q][e.p] < e.TS[e.p]$ **do**
    append e to L'_j
  end
  append L'_j to LL'
end
send LL' to q
send min_TS to q

Algorithm 2: $sendLog(q)$

The CORBA Portable Interceptor standard defines two standard interfaces "ClientRequestInterceptor" and "ServerRequestInterceptor".[10] In the interceptor, the following interception points were used to weave the update log propagation system into the system:

Client or Server	Method	Details
Client	send_request	Queries request information and modifies request service context.
Client	receive_reply	Queries reply information after server has completed call.
Server	receive_request	Queries request information and modifies reply service context.
Server	send_reply	Queries reply information after target operation excution and before reply is sent to client. Modifies reply service context.

On the client side, we implemented an client interceptor *send_request* method that uses the modified *sendLog* algorithm listed above and creates propagation logs for all possible communication partners. After the propagation logs are created, they are serialized, along with the *min_TS* matrix timestamp into the request context. The server side interceptor implements the *receive_reply* method and unmarshalls the logs, throws away the logs which do not correspond to its node id, and utilizes the *receiveReply* algorithm to incorporate the updates into its event log. It also places a context hint into the ORB-supplied reply context identifying the client node, so that the more optimal *sendLog* algorithm can be utilized on during the reply. After the operation has been invoked, the *send_reply* method reads the context hint out of the ORB-supplied reply context (which is the serialized id of the client node) and builds an event propagation log using the standard algorithm and serializes this along with the server's *min_TS* matrix timestamp into the reply context. The client side interceptor *receive_reply* method retrieves the *min_TS* and the update propagation log from the reply context and uses the *receiveReply* algorithm to incorporated the updates into its event log.

3.3. PROFILING THE SYSTEMS

One of the most difficult challenges facing us during the course of this experiment was obtaining consistent profiling results from the application without materially affecting the runtime of the system, as the PDLS system was found to be highly

latency-sensitive. In preliminary tests, it was noticed that a network condition was causing an increase in the latency associated with sending small messages. Due to the synchronization feature of the network rendezvous, any small additions in latency can aggregate and cause large delays at the system level. Therefore, a low-overhead call counting and profiling library, inspired by the Solaris-based DTrace system[18], was developed. This library, called *Simpletrace* in the implementation of the system, was designed to impose a minimum amount of overhead on the system under observation. In order to ensure a constant of amount of overhead, the Simpletrace was developed in C/C++ and a Python module wrapper was created for it. This ensured that the same amount of overhead was imposed on each system, allowing us to directly compare runtime results.

As mentioned before, the *Simpletrace* library measures only call counts and total call times. In the interest of achieving a low-overhead, minimum footprint profiling toolkit, stack traces are not obtained during the profiling calls. This limitation limits the applicability of the *Simpletrace* library to cases where there is *a priori* knowledge of the call-tree and logic flow. When a method is entered, the *logMethodEnter* function is called passing the name of the class and the method called as a static string of the format "ClassName.methodName". On method exit, the *logMethodExit* function is called, again passing the name of the class and the method called as a static string formatted "ClassName.methodName". When a function is entered, the *logMethodEnter* function is called passing the function called as a static string of the format "functionName". On method exit, the *logMethodExit* function is called, again passing the name of the class and the method called as a static string formatted "functionName".

When the application has finished executing, the *Simpletrace* library compiles statistics for each method and function which it encountered during the running of the system. The call count, total time, and average time for each method and function are printed to standard error for each thread. All results are comma-separated value formatted, for easy importation into Microsoft Excel. This was found to greatly reduce the amount of time required to analyze the results.

4. MODEL PROBLEMS

To study the performance of the of the PDLS system and the TAO-based Portable Interceptors, we implemented two model problems. The first model problem is the classic Bounded-Buffer Problem, which was optimal for testing the system with a lower-number of nodes, and could be completely contained on one host. The second model problem is the BOOTS case study[13, 12], as implemented by the CCSP system[8, 16]. However in the system implemented for the model problem, there is no notion of history sanitization[16], as this is not relevant or necessary to profile the system. The BOOTS system, as implemented in the CCSP system, is an excellent example of a loosely-connected, large node set distributed system. In the model problems that follow, each node is a separate processor, communicating with other nodes via CORBA or PDLS.

4.1. BOUNDED-BUFFER PROBLEM

The Simple Bounded-Buffer Problem is a well-known problem in the field of Computer Science. There are three nodes in the system, a Producer, a Consumer, and a Buffer. The Producer node produces items, which are transmitted to the Buffer in a "GIVE" operation. The Consumer Node requests items from the Buffer using the "TAKE" operation. The Buffer takes items transmitted by the Producer and stores them in a bounded FIFO queue of items. For the purposes of the later experiments, one may assume that the bound placed on the queue is that it make contain no more than ten items and no less than zero. When the Buffer detects that the queue is full, it stops servicing "GIVE" requests and only communicates with the Consumer node. Likewise, when the Buffer detects that no items are left in the queue to consume, it stops servicing "TAKE" requests and communicates exclusively with the Producer node. When the queue is not either full or empty, the Buffer node will service requests from either the Producer or the Consumer.

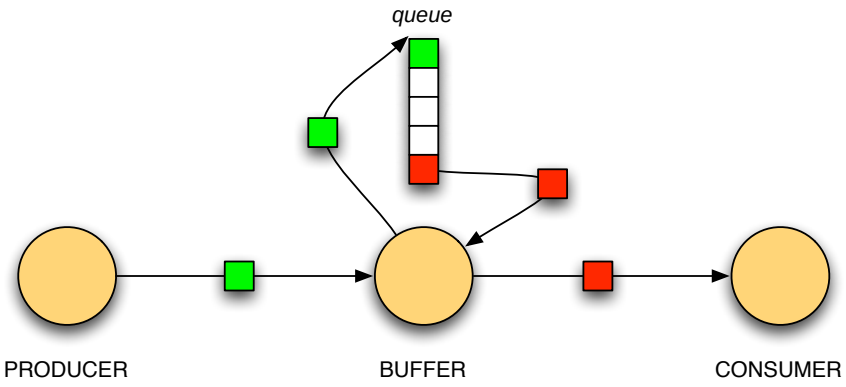

Figure 4.1. Bounded Buffer Problem

4.2. BOOTS2 SYSTEM

The Boots System simulates the movement of footwear orders through an ordering and distribution system. Orders can be labeled with differing security levels and certain security constraints are followed which makes the system interesting due to the complexity and volume of the message traffic.[16] An order consists of a destination for the shipment, a quantity of boots to be moved, and the purpose for moving the boots. Orders have sensitivity levels of either high or low.

The BOOTS System has the following classifications of node:

- HeadQuarters (HQ) - The node where the orders originate.
- Stock-cell (SH,SL) - The nodes which decide the type of boots required based on the the order's purpose. The nodes also decide a source for the boots. Orders with high security classifications are routed to the SH node and orders with a low security classification are routed to the SL node.
- Stock-records (SR) - Coordinates with SH or SL to decide the source of the boots which will fulfill the order.
- Security Officer (SO) - If an order is over-classified, the HQ node will send it to the SO node, where it will be regraded and set to the SL node. The SO node also inspects senders and receivers of audited messages in the Auditing subsystem.

- Movement (MV) - The MV node is messaged by either SL or SH with the number, source, and destination of the boots in the order. The node calculates the number of trucks necessary for the shipment of the boots, and propagates this information, along with the source and destination attributes of the shipment, to the Transport node.
- Transport (TRP) - The TRP node is messaged by the MV cell with the source and destination of a boots order, along with a number of trucks needed to move a quantity of boots. The TRP node checks with the Transport Records node to decide which trucks to utilize in order to fulfill the boots order.
- Transport Records (TRR) - The TRR node is queried by the TRR node when it is deciding which trucks to utilize in order to fulfill the boots order.
- Auditor Buffer (ABF) - Buffers auditing messages from the rest of the system to the Auditor node.
- Auditor (AUD) - Audits security messages held by the ABF node.
- Operator (OP) - Participates in the the archiving of an audit trail.
- Archive (ARH) - The ARH node receives security messages from the AUD node and archives them.

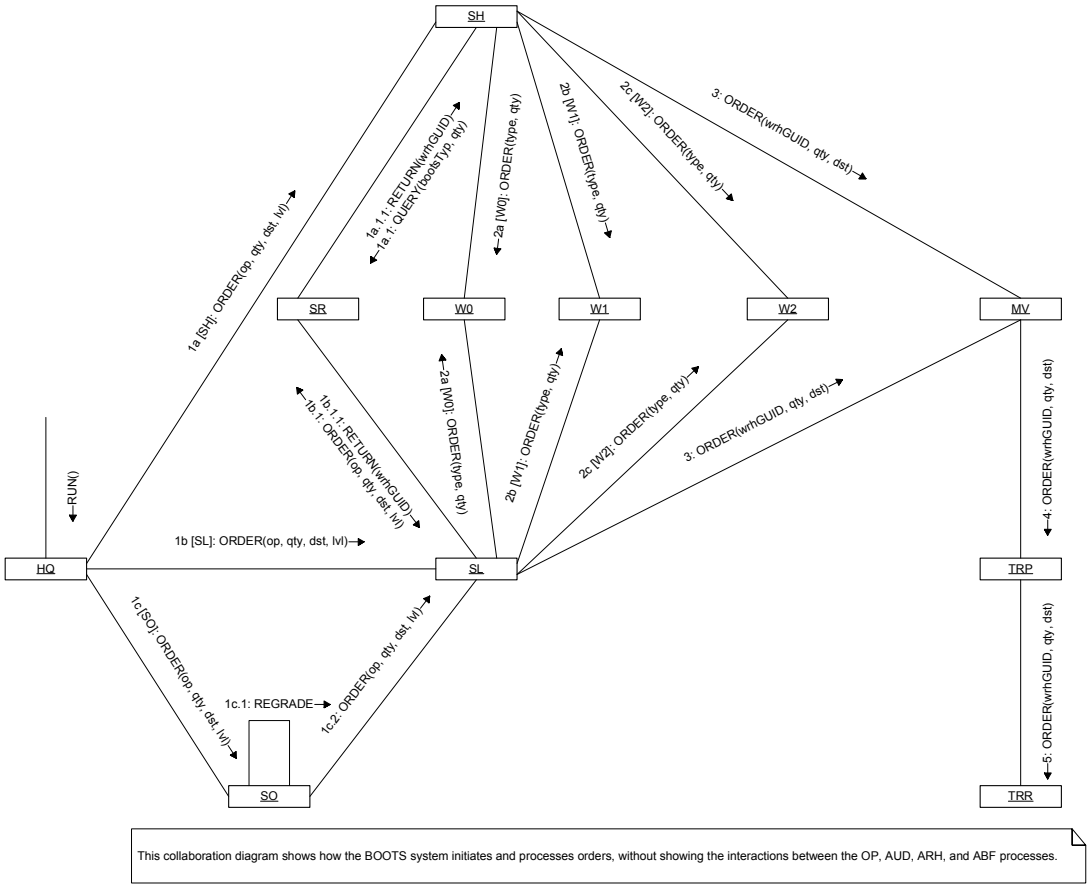

Figure 4.2. UML Collaboration Diagram Showing the Processing of An Order within the BOOTS2 System

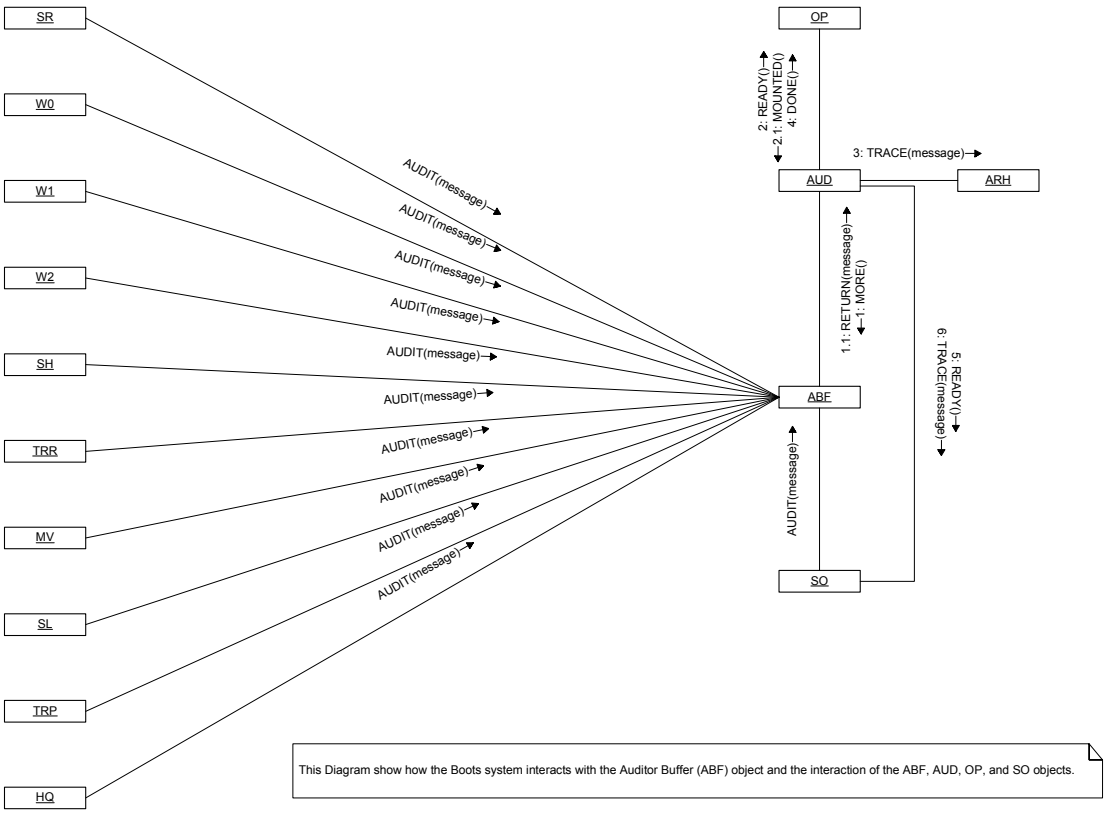

Figure 4.3. UML Collaboration Diagram Showing the Interaction of the BOOTS2 Nodes with the Auditor, Auditor Buffer, Operator, and Security Officer Processes

5. RESULTS

Four distributed systems were created to test the PDLS and CORBA middleware frameworks. First, we created a CORBA-based version of the Bounded Buffer problem and a PDLS-based version of the Bounded Buffer problem. In order to see the overhead of each system, the runtime of each system was modeled by running it for 10000 iterations and changing the message size for each test, starting with a message size of 256 bytes. In each test, the message size was equal to the message size of the previous test multiplied by two. Using the Simpletrace library, we simultaneously profiled the overhead of each part of the system, grouping the various time components into either the "Transport", "Middleware", or "Application" time categories. The "Transport" category was assigned the network I/O time. w/o any wait time (time typically spent in a select loop) The "Middleware" time category was assigned the wait time, event propagation and incorporation time, marshalling and unmarshalling time, and any other overhead assigned to the middlware in test. The "Application" category receives the balance of the time unassigned to the "Transport" or "Middleware" categories. Using this profiling data, we were able to determine the overhead imposed by the middleware under test. For the Bounded-Buffer problem, we also tracked the number of bytes sent and received by each node in the system. Using this data, we created a graph and used linear regression to model the runtime of the system using a version of the linear model of point to point communication. As the message payload size of the BOOTS system was driven by each individual node, this modelling was not performed for the large-scale BOOTS system test.

5.1. LINEAR MODEL OF POINT-TO-POINT COMMUNICATION

The linear model of point-to-point communication is used to model the communications between two nodes of a distributed system. The parameters are as follows:

- t - total time
- t_s - startup time
- t_b - time needed to send one byte of data in the message
- n - number of bytes

$$t = t_s + t_b * n \tag{1}$$

In our experiments, we used the total runtime of the system as our t value, we tracked the total number of bytes sent in the system and used this for our n value, and used linear regression to solve for t_s and t_b, using our set of results garnered from the test runs. Again, each test used a different message payload size, starting from 256 bytes, and doubling with each test run until a message payload size of 8192 bytes was reached.

5.2. BOUNDED BUFFER

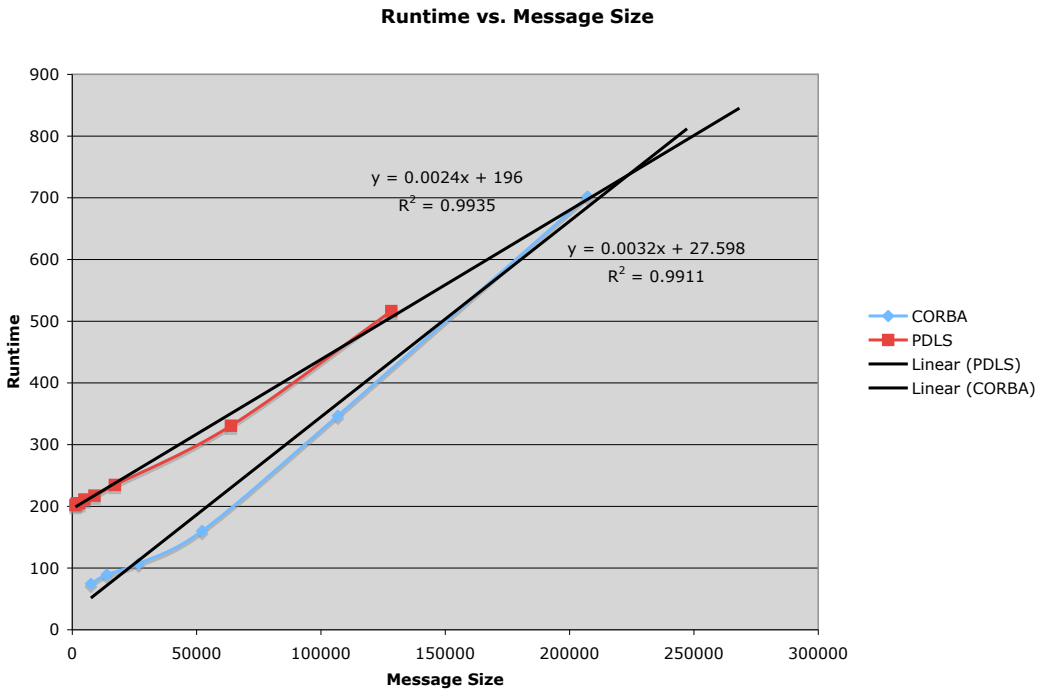

Figure 5.1. System Runtime vs. Message Size

This graph shows that the CORBA-based system outperforms the PDLS system until the message size reaches 210, 502 bytes. The message size is dependent on the number of the nodes in the system, payload size, marshalling efficiency, and algorithm used to create the history update. This is why the efficiency of the algorithm used in the PDLS-based system is critical for overall system efficiency.

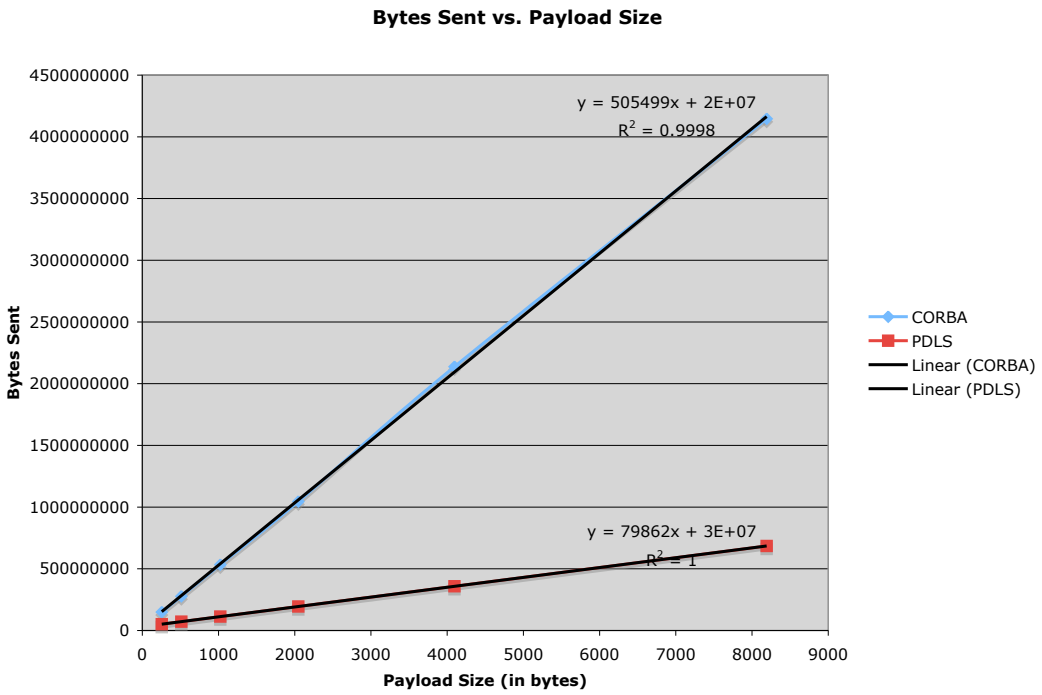

Figure 5.2. Total Traffic (Bytes Sent) vs. Payload Size

The inefficiency of the CORBA-based solution is also shown here, with the increase in the total traffic (in bytes) increasing at a rate which is approximately 6x that of PDLS. The relative inefficiency of IIOP vs. the pickle packet may also contribute to this result.

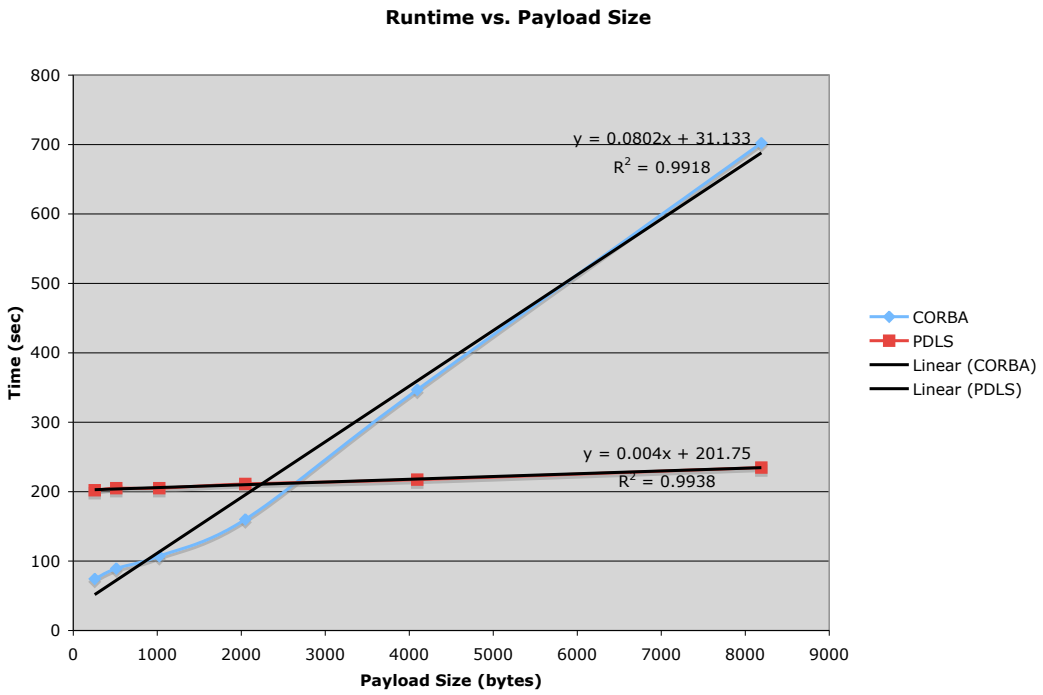

Figure 5.3. System Runtime vs. Payload Size

This graph again highlights inefficiency in the CORBA system. The intersection point of the trendlines in this graph occurs at message payload size 2239 bytes.

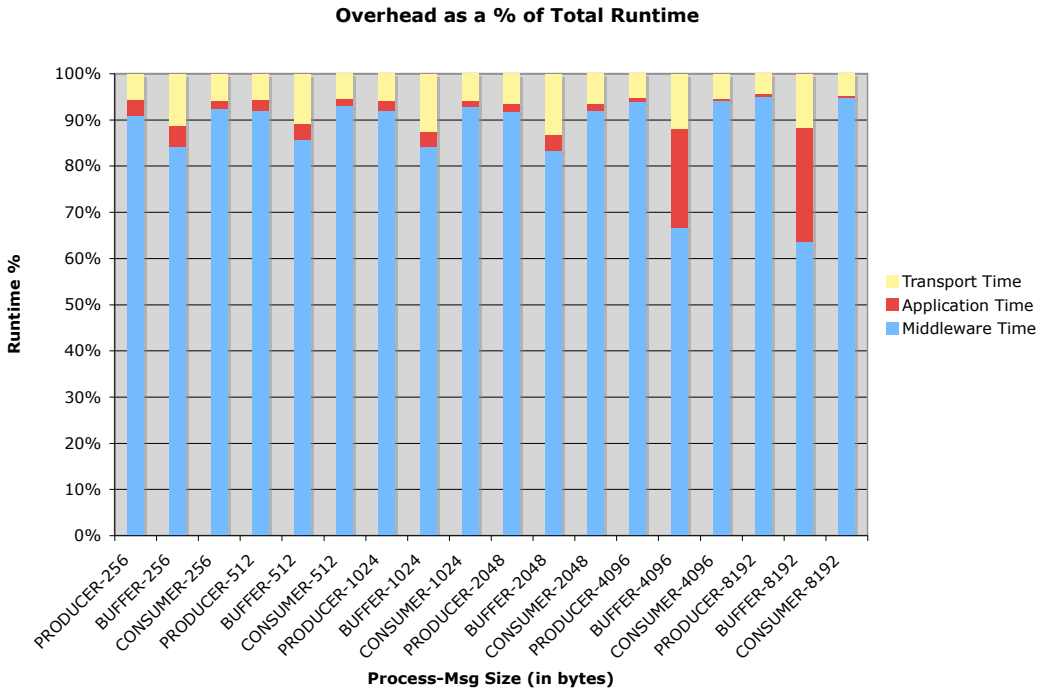

Figure 5.4. Overhead Imposed by the CORBA Middleware on the Bounded Buffer Problem

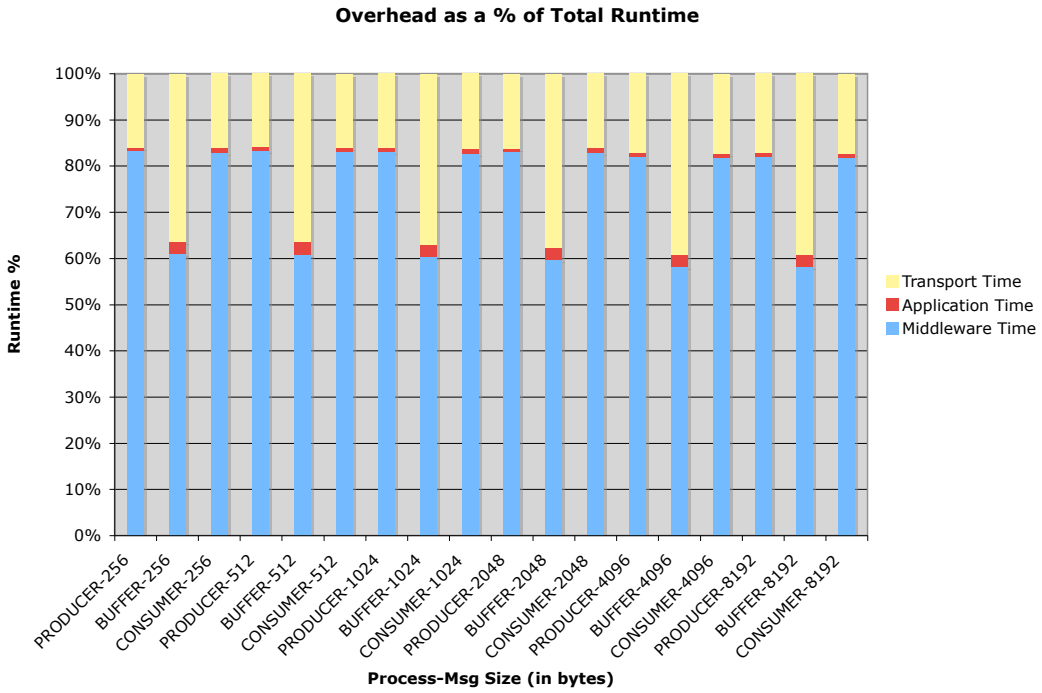

Figure 5.5. Overhead Imposed by the PDLS Middleware on the Bounded Buffer Problem

The previous two graphs show that the PDLS solution imposes much less overhead (5-10% less) than the CORBA-based solution for the PRODUCER and CONSUMER nodes. Since wait time is charged to the Middleware Time category, the effect is much more pronounced in the BUFFER node, which spends less of its time waiting for messages, as the PRODUCER node must wait for the BUFFER while it is communicating with the CONSUMER node and vice-versa. In the case of the BUFFER node, the difference in overhead is approximately 25%-30%.

5.3. BOOTS SYSTEM

	TRR		MV		HQ	
	time (μs)	time %	time (μs)	time %	time (μs)	time %
Application	122187	0.025%	359422	0.072%	4952360	1.002%
Transport	1804083	0.363%	21831720	4.395%	1764340	0.357%
Middleware	21494262	99.612%	116074679	95.532%	487720371	98.642%

Table 5.1. CORBA Results

	TRR		MV		HQ	
	time (μs)	time %	time (μs)	time %	time (μs)	time %
Application	449472	0.16%	902571	0.31%	702387	0.24%
Transport	7195336	2.49%	15092049	5.17%	10180044	3.50%
Middleware	281393175	97.36%	275966245	94.52%	280339237	96.26%

Table 5.2. PDLS Results

The tables above show the results of the BOOTS system implementation using both the CORBA and PDLS frameworks. As expected, the system runtime and overhead of the PDLS system is much lower than the overhead of the CORBA framework, due to the higher overhead of the update log propagation implementation. This effect was also seen in the small-scale (Bounded Buffer Problem) tests. The BOOTS system has 15 nodes, but for this test, the Auditing subsystem (ARH, OP, ABF, and AUD) was disabled due to performance issues in the scheduling component of the PDLS

system. A representative subset of the remaining nodes (HQ, TRR, and MV) were selected and profiled, giving the results seen above.

5.4. ISSUES FACED DURING IMPLEMENTATION

5.4.1. Latency Sensitivity. During the implementation of the PDLS system, we noticed that with very small message payload sizes (less than 2048 bytes), the "Middleware Time" category spiked and the overall runtime of the system was adversely affected. This observed behavior was due to the synchronization aspect of the underlying rendezvous primitive, which waits for the completion of the bidirectional data transfer. The operating system was attempting to coalesce small messages into larger messages in the network buffers, which in turn was adding a delay to each data transfer and increasing the latency of the rendezvous. These delays were accumulating to the point where the overall system performance was negatively impacted. After much profiling, debugging, and research, it was noticed that the CORBA-based system disabled "Nagle's Algorithm"[17] for small messages, alleviating this problem by disabling the message coalescing behavior. When Nagle's Algorithm was disabled in the PDLS system, the transmission time of small messages in the Transport Category greatly decreased and the overall system runtime was significantly reduced.

5.4.2. Pervasive Concurrency - Race Conditions. The development of the PDLS system was fraught with peril. Threading, while useful, opened up many conditions where races, deadlocks, and subtle logic bugs created multi-day debugging scenarios and difficult to replicate failure situations. This greatly complicated the debugging and lengthened the development time of the PDLS system.

5.4.3. Python Interpreter Concurrency. Another issue encountered in development due to a known issue in the Python Interpreter and was exacerbated when the PDLS system was used on multiple-processor hosts. The issue centers around the need for the Python interpreter to preserve state which is common across all threads. Locking the entire interpreter makes it impossible for concurrent processing to occur within one context space. This lock is known as the Global Interpreter Lock, or the GIL and is released on I/O and at a set interval set in the "sys" module as the "syscheckinterval". In the PDLS interpreter, it was found that manipulating this value had little to no effect on the overhead imposed by the Middleware or

Transport layers. However, by creating a C module for the TagDict module utilized by the RendezvousService module, which allowed a thread waiting on a key to give up the GIL, it was theorized that the concurrency level of the system could be greatly increased and the performance improved on multiple core systems. However, those theories were not tested and the tests are reserved for future work.

6. CONCLUSIONS

We created a lightweight middleware that utilized the power of a dynamic language to make the implementation of secure and fault-tolerant applications more straightforward and less involved, without sacrificing an undue amount of performance and concurrency. The largest disappointment encountered was the due to the runtime's lack of concurrency (i.e. the GIL), which limits performance on hosts with multiple processors or processor cores.

In comparison to a similar solution implemented using CORBA, PDLS is the clear winner, due to one of the design decisions made early on in CORBA which limits the transparency and visibility of the node which will service another node's request. The TAO technology addition (available in the Beta version of TAO but not in the Generally Available release at the time of writing) which will allow interceptors to know more about the endpoint (IP address and TCP port), should allow a future researcher to ameliorate this issue and bring the efficiency of a CORBA-based solution into line with a PDLS one. Unfortunately, for now, this limitation severely limits the scalability of a CORBA-solution, due to the need to create an update log for each possible node in the system for at least request part of the RPC.

A look at the code listing in Appendix A for the Bounded Buffer problem will highlight the simplicity of the PDLS solution vs. the CORBA solution. The PDLS-framework is ideal for the implementation of fault-tolerant and security applications without having to distract the implementor with marshalling (type), memory-managment, or language (IDL) concerns.

7. FUTURE WORK

The PDLS system is relatively new, and as such could use improvements in the protocol used for message passing and in the framework itself. In the following subsections, we examine and suggest areas for improvement, as well as suggesting some applications for the middleware.

7.1. PROTOCOL IMPROVEMENTS

The PicklePacket described in the previous sections is an all purpose solution for serializing an object graph to a bytestream. While this generic system works very well and appears to perform well, it is possible that a more optimal solution, one which does not sacrifice the design goal of protecting the application developer from the nuances of marshalling and unmarshalling data, might exist and be superior to the generic system of pickling Python object graphs. An efficiency comparison, in which a known data structure and was marshalled, unmarshalled, and profiled might be advantageous.

7.2. FRAMEWORK IMPROVEMENTS

The auditing subsystem of the BOOTS was disabled in the above experiment due to poor performance seen in the ABF process, which was processing events from a large number of nodes in the system. The poor performance was attributed to the lack of intelligent scheduling in the RendezvousService component of the PDLS system, as it merely randomizes the list of tags it waits for and takes the first available tag, allowing a persistent process to monopolize the conversation, starving the other processes. A "pluggable" system of schedulers for the RendezvousService would be advantageous, as it would allow the application implementor to select the scheduler best suited to the solution's requirements.

The PDLS system is perfectly suited to a static set of nodes, where nodes do not enter or leave the system during a long running distributed system. Unfortunately, in reality, this is very rarely the case. The applicability of PDLS is limited due to the inability of the middleware to dynamically add and remove nodes at runtime.

The addition of this functionality would allow the middleware to be used in mobile applications and in Peer-to-Peer file sharing and indexing applications.

7.3. APPLICATIONS

The PDLS system would be ideal for creating applications to facilitate research into security system, such as intrusion detection research using immune system inspired detectors on systems such as the BOOTS system.[15] The PDLS middleware framework supports logging of all update logs, and these update logs (or "traces") can be used as input data for these security systems.

We suggest that PDLS would be ideal for certain fault-tolerant applications, as a connected graph of nodes is not necessary to propagate updates to the global state throughout the system. This makes the system robust with respect to lost connections, corrupted data, compromised systems, and non-responsive nodes.

APPENDIX A

BUFFER Implementation

This appendix shows the source code of the BUFFER implementation utilizing the PDLS and CORBA middleware frameworks. In the case of PDLS you see the entire application, with the addition of the PDLS framework, it is self-contained and ready to run. In the case of the CORBA-based BUFFER implementation, I show only the C++ class implementing the BUFFER functionality, as the driver ("main.cpp") is mostly TAO CORBA C++ boilerplate code.

Listing 1. PDLS Implementation of BUFFER Node

```
from PDLS import *
from PDLS.TaggedObject import TaggedObject
from PDLSPids import *
from PDLS.Loggers import TextLogger
from tracesupport import traceit_method, traceit_func

import time, sys, os

# PIDS
# 0 - Buffer
# 1 - Consumer
# 2 - Producer

# Main function

numItems = 10000
stdMsg = ''.join(['X' for i in range(int(sys.argv[1]))])

@traceit_func
def bufferKernel(es):
  # Setup the bounded buffer
  bBuffer = []
  numProduced = 0
  numConsumed = 0

  for i in range (numItems * 2):
```

```
# If we have room, accept a rendezvous from either the
# producer or the consumer

if ( (len(bBuffer) > 0) and (len(bBuffer) < 10) ):
  tObjGive = TaggedObject(guid = pidProducer,
    tag = 'GIVE', containedObj = stdMsg)
  tObjTake = TaggedObject(guid = pidConsumer,
    tag = 'TAKE', containedObj = bBuffer[0])
  # Do the guarded recv
  rObj = es.selectEvents( [ tObjGive,tObjTake ] )
  tag = rObj.getTag()
  if (tag == 'GIVE'):
    # We rendezvous'd with the producer
    bBuffer.append(rObj.getObject())
    numProduced += 1
  elif (tag == 'TAKE'):
    # We rendezvous'd with the consumer
    bBuffer.pop(0)
    numConsumed += 1
  else:
    raise RuntimeError, "Bad TAG %s" % tag
elif (len(bBuffer) == 0):
  # Only rendezvous with the producer
  tObj = TaggedObject(guid = pidProducer,
    tag = 'GIVE')
  rObj = es.recvEvent( tObj )
  bBuffer.append(rObj.getObject())
  numProduced += 1
elif (len(bBuffer) == 10):
  # Only rendezvous with the consumer
  tObj = TaggedObject(guid = pidConsumer,
    tag  = 'TAKE', containedObj = bBuffer.pop(0))
  rObj = es.sendEvent( tObj )
  numConsumed += 1
```

```
    i+=1


def mainLoop(es):
  time.sleep(10)

  bufferKernel(es)
  print "bufferKernel exited."
  time.sleep(10)
  es.shutdown()

def main():
  nameTable = NameTable.deserializeFromFile(
    os.path.expanduser('~/research/nametables/bbuffer.xml'))
  es = EventService.EventService(pidBuffer, 3, None,
      nameTable, 1, TextLogger('/tmp/PDLSBuffer.elog'),
      True, 2, mainLoopFunc=mainLoop,
      seqNumStart=pidBuffer * 1000000)
  es.listen()

###

if __name__ == '__main__':
  main()
```

Listing 2. CORBA Implementation of BUFFER Node - IDL

```
module BBUF
{
  interface BUFFER
  {
    void GIVE(in string item);
    string TAKE();
  };
```

```
};
```

Listing 3. CORBA Implementation of BUFFER Node - C++ Class Header

```
#include "BUFFERS.h"
#include "LazyDB.h"

class BUFFER_i : public POA_BBUF::BUFFER
{
public:
  BUFFER_i(const std::string& _stdMsg);

  ~BUFFER_i();

  void orb (CORBA::ORB_ptr o);
  // Set the ORB pointer.

  void poa (PortableServer::POA_ptr poa);
  // Set the POA pointer.

  void set_orb_manager (TAO_ORB_Manager *orb_manager);
  // Set the ORB Manager.

  TAO_ORB_Manager *orb_manager_;
  // The ORB manager.

  virtual void GIVE (
    const char * item
    )
    ACE_THROW_SPEC ((
    ::CORBA::SystemException
    ));

  virtual char * TAKE ()
    ACE_THROW_SPEC ((
```

```
    ::CORBA::SystemException
    ));

  void orbShutdownCheck();

private:
  CORBA::ORB_var orb_;
  // ORB pointer.


  PortableServer::POA_ptr poa_;
  // POA pointer.

  ACE_Thread_Mutex listMutex;
  ACE_Semaphore itemsAvailSemaphore;
  ACE_Semaphore spaceAvailSemaphore;
  ACE_Atomic_Op<ACE_Thread_Mutex,int> numEvents;


  std::list<char *> stringBuffer;
  std::string stdMsg;

  ACE_UNIMPLEMENTED_FUNC (void operator= (const BUFFER_i &))
};
```

Listing 4. CORBA Implementation of BUFFER Node - C++ Class Implementation

```
#include "StdAfx.h"
#include "BUFFER_i.h"
#include "BBUFPids.h"
#include "EventLibTypes.h"
#include "Simpletrace.h"
#include "CORBAOBJ_factory.h"

#define MAX_BUFFER_SIZE 10
```

```
#define NUM_MSGS 10000 * 2

BUFFER_i::BUFFER_i(const std::string& _stdMsg) :
  itemsAvailSemaphore(0),
  spaceAvailSemaphore(MAX_BUFFER_SIZE),
  stdMsg(_stdMsg), numEvents(0)
{
  // no-op
}

BUFFER_i::~BUFFER_i()
{
  // no-op
}

void
BUFFER_i::orb (CORBA::ORB_ptr o)
{
  this->orb_ = CORBA::ORB::_duplicate (o);
}

void
BUFFER_i::poa (PortableServer::POA_ptr poa)
{
  this->poa_ = poa;
}

void
BUFFER_i::set_orb_manager (TAO_ORB_Manager *orb_manager)
{
  this->orb_manager_ = orb_manager;
}

void
```

```
BUFFER_i::GIVE(const char * item)
ACE_THROW_SPEC ((::CORBA::SystemException))
{
  logMethodEnter("BUFFER_i.GIVE");

  spaceAvailSemaphore.acquire();

  ACE_Guard<ACE_Thread_Mutex> guard(listMutex);
  char *tmpString = new char[strlen(item) + 1];
  strcpy(tmpString, item);
  this->stringBuffer.push_back(tmpString);
  guard.release();
  itemsAvailSemaphore.release();

  eventLogSingleton->lockForUpdate();
  eventLogSingleton->
    performUpdate(std::string("(GIVE,") +
    stdMsg + std::string(")") );
  eventLogSingleton->unlockForUpdate();

  numEvents++;
  orbShutdownCheck();

  logMethodExit("BUFFER_i.GIVE");
}

char *
BUFFER_i::TAKE()
ACE_THROW_SPEC ((::CORBA::SystemException))
{
  logMethodEnter("BUFFER_i.TAKE");

  itemsAvailSemaphore.acquire();
  ACE_Guard<ACE_Thread_Mutex> guard(listMutex);
```

```
  char *tmpString = this->stringBuffer.front();
  this->stringBuffer.pop_front();
  guard.release();
  spaceAvailSemaphore.release();

  eventLogSingleton->lockForUpdate();
  eventLogSingleton->
    performUpdate(std::string("(TAKE,") +
    stdMsg + std::string(")") );
  eventLogSingleton->unlockForUpdate();

  numEvents++;
  orbShutdownCheck();

  logMethodExit("BUFFER_i.TAKE");

  return tmpString;
}

void
BUFFER_i::orbShutdownCheck()
{
  logMethodEnter("BUFFER_i.orbShutdownCheck");
  if (numEvents >= NUM_MSGS)
  {
    // Shutdown the orb, wait for all
    // events to complete first.
    std::cerr << "BUFFER is shutting down"
      << std::endl;
    (*orbSingleton)->shutdown(true);
  }
  logMethodExit("BUFFER_i.orbShutdownCheck");
}
```

APPENDIX B

PDLS User Manual

This Appendix constitutes the PDLS User Manual. The PDLS messaging API is exposed via the *EventService* class. Data is contained in an instance of the *TaggedObject* class. The *EventService* requires a properly initialized *NameTable* to be initialized with the GUIDs, TCP ports, and IP addresses of all of the other nodes in the distributed system, as shown below in the example *NameTable* XML file.

B.1. USAGE EXAMPLE

The example below sets up a *NameTable* from a serialized xml file and starts the EventService. When the EventService finishes initializing the listening thread, used for the receiving of *TaggedObjects* from remote nodes, it call the *mainLoop* function, which proceeds to demonstrate the use of the select, receive, and send primitives of the PDLS middleware.

Listing 5. Example NameTable XML File

```
<?xml version='1.0' encoding='UTF-8' ?>
<nametable>
    <!-- this entry tells the NameTable that
         the node with GUID = '0'
         is running on the host 'blade3.cs.umr.edu'
         at TCP port '26788' revision is currently out,
         but will act as a timestamp for
         a future merge/update algorithm
    -->
    <entry ip='blade3.cs.umr.edu' guid='0' port='26788'
      revision='0' />

    <!-- the following entries are in the same
         form as the first entry
         and define the rest of the nodes
         in the NameTable
    -->
    <entry ip='blade4.cs.umr.edu' guid='1' port='26789'
      revision='0' />
```

```
    <entry ip='blade5.cs.umr.edu' guid='2' port='26790'
      revision='0' />
</nametable>
```

Listing 6. Example PDLS Python Script

```
from PDLS import *
from PDLS.TaggedObject import TaggedObject
from PDLSPids import *
from PDLS.Loggers import TextLogger

# This function is called by the EventService,
# after it has completed setting up the
# listening thread.
def mainLoop(es):

  # setup a pair of test objects for the select
  tObjGive = TaggedObject(guid = pidProducer, tag = 'GIVE',
        containedObj = stdMsg)
  tObjTake = TaggedObject(guid = pidConsumer, tag = 'TAKE',
    containedObj = 'some␣test␣obj')

  # This shows a multiple-receive (a select)
  rObj = es.selectEvents( [ tObjGive,tObjTake ] )
  tag = rObj.getTag()

  # This shows a single-receive
  # We'll wait until the remote node (GUID = 0)
  # contacts us with a Tagged object with the
  # tag 'GIVE'
  tObj = TaggedObject(guid = 0, tag = 'GIVE')
  rObj = es.recvEvent( tObj )
  # rObj is the TaggedObject sent by the remote node and
  # exchanged during the rendezvous
```

```
    # This shows a single-send
    # We'll send this TaggedObject to the remote node (GUID = 2)
    # and wait for a reply.
    tObj = TaggedObject(guid = 2, tag  = 'TAKE',
          containedObj = 'some test payload')
    rObj = es.sendEvent( tObj )
    # rObj is the TaggedObject returned by the remote node and
    # exchanged during the rendezvous

    # shutdown the event service, kill the listening thread
    es.shutdown()

def main():
    # deserialize a previously created nametable from the xml
    # file at $HOME/research/nametables/testnametable.xml
    # this will allow us to resolve GUIDs to TCP ports and IP
    # addresses
    nameTable = NameTable.deserializeFromFile(
      os.path.expanduser(
        '~/research/nametables/testnametable.xml'
      ))

    # setup a TextLogger to log the events to
    # '/tmp/testnode.log'
    logger = TextLogger('/tmp/PDLSBuffer.elog')

    # setup the EventService instance, the instance should
    # start the 'mainLoop' after it has properly
    # initialized the listening thread
    es = EventService.EventService(1, # our node GUID
      3, # total nodes
      None, # TCP port to listen to (this is default, look
            # up in NameTable)
      nameTable, # the previously initialized nametable
```

```
        1, # turn on history clipping (should always do this
           # unless you have a _good_ reason to disable this)
        logger, # pass the text logger
        True, # keep alive connections (should always do this
              # unless you have a _good_ reason to disable
              # this)
        2, # expected number of nodes we're communicating
           # with (optimization of thread pool)
        mainLoopFunc=mainLoop # function to call after
                              # we initialize the
                              # listening thread
        )

    # tell the EventService to start the listening thread,
    # which will then start the mainLoop() function, passing
    # itself as the first parameter
    es.listen()

###

if __name__ == '__main__':
    # Execute the main() function
    main()
```

B.2. API REFERENCE

TaggedObject Class

__init__(*self*, *guid*, *tag*, *containedObj*=None)

```
Initialize a TaggedObject.

Parameters:

    'guid' - GUID of destination node
    'tag'  - data tag associated with the 'TaggedObject' instance
    'containedObj' - payload object
```

__eq__(*self*, *other*)

```
Test for equality between this instance and
another instance of a 'TaggedObject'

Parameters:

    'other' - instance of 'TaggedObject' to test against for equality

Returns True if equal, False if not equal.
```

__str__(*self*)

Returns the string representation of the 'TaggedObject'.

getGUID(*self*)

Returns the GUID of the destination node.

getObject(*self*)

Returns the payload object.

getSrcGUID(*self*)

Returns the GUID of the source node

getTag(*self*)

Returns the data tag.

setSrcGUID(*self*, *srcGUID*)

```
INTERNAL
Sets the GUID of the source node.

Parameters:

    'srcGUID' - GUID of the source node

Note:

    Should only ever be called by the RendezvousService.
```

EventService Class

__init__(*self*, *pid*, *numPids*, *port*=None, *nameTable*=None, *clipping*=1, *logger*=None, *keepAliveConnections*=False, *numExpectedPeers*=5, *mainLoopFunc*=None, *seqNumStart*=0)

```
Initialize an EventService object.

Parameters:

    'pid' -- pid of the process in the system
    'numPids' -- number of processes in the system
    'port' -- (optional) TCP/IP port number to listen on
    'nameTable' -- (optional) 'NameTable' object to use in
        this object
    'clipping' -- (optional) 1 if events should be clipped
        from the history, 0
    'logger' -- (optional) logger to log events
    'keepAliveConnections' -- (optional) passed to NetworkService
    'numExpectedPeers' -- (optional) passed to NetworkService
    'mainLoopFunc' -- (optional) passed to NetworkService
    'seqNumStart' -- (optional) passed to NetworkService
    if they should not
```

buildPropLog(*self*, *q*, *tempL*)

```
INTERNAL
Build a log of events to propagate to pid 'q'.

Parameters:

    'q' -- pid of the process we are building an event
    history propagation log for
    'tempL' -- the log used as the source log

Returns:

    A propagation log to send to 'q'.
```

dumpLog(*self*)

```
     Dump our log out to stdout.
```

getLogger(*self*)

```
     Returns the logger for the 'EventService'
```

getRendezvousService(*self*)

```
Returns the embedded 'RendezvousService' object associated
with this object.

Returns:

    The 'RendezvousService' object associated with this object.
```

listen(*self*)

```
Begin listening for events.
```

receiveLog(*self*, *p*, *Lprime*, *min_TSprime*)

```
INTERNAL
Merge 'Lprime' (the propagated history from 'p') with our
L (event history). Also, update our 'minTS' members with the max
of 'minTS' and 'min_TSprime'.

Parameters:

    'p' -- pid of the process we received the propagated history from
    'Lprime' -- the propagated history from 'p'
    'min_TSprime' -- minimum 'IntFixedVectorTS' object from 'p'
```

recvEvent(*self*, *tObj*, *copyObjectFlag*=0, *timeout*=0)

```
Rendezvous at tag with srcPid, passing our event history
for the object and our matrix timestamp.

Parameters:

    'tObj' -- TaggedObject to be exchanged with the sending
        process.
    'copyObjectFlag' -- 0 if the object contained in tObj
                        should not be deep-copied
                        1 if the object contained in tObj
                        should be deep-copied
    'timeout' -- unused

Returns:

    The 'TaggedObject' sent from the sending process.
```

selectEvents(*self*, *tObjList*, *copyObjectsFlag*=0, *timeout*=0)

```
Inspired by Ada's select.
Implements a guarded recv with the pids and tags
specified in the tObjList.

Parameters:
    'tObjList' -- a list of TaggedObjects used for specifying the
    pids, tags, and objects used in the redezvous
    'copyObjectsFlag' -- 0 if the objects contained in tObjList
                         should not be deep-copied
                         1 if the objects contained in tObjList
                         should be deep-copied
    'timeout' -- unused

Returns:
    The 'TaggedObject' sent from the sending process.
```

sendEvent(*self*, *tObj*, *copyObjectFlag*=0, *timeout*=0)

```
Rendezvous at tag with destPid, passing the tObj object
and the matrix timestamp.

Parameters:

    'tObj' -- 'TaggedObject' which is to be sent
    'copyObjectFlag' -- 0 if the object contained in tObj
                        should not be deep-copied
                        1 if the object contained in tObj
                        should be deep-copied
    'timeout' -- unused
```

setLogger(*self*, *logger*)

```
Sets the logger for the 'EventService'
```

shutdown(*self*)

```
Shuts down the internal 'NetworkService' instance.
```

NameTableEntry Class

__init__(*self*, *guid*='', *ip*='*', *port*=0, *revision*=0)

```
Initializes a 'NameTableEntry' object.

Parameters:

    'guid' -- guid (string/int) representing a process

    'ip' -- TCP/IP address of a process

    'port' -- TCP/IP of the port of a process

    'revision' -- versioning of the 'NameTableEntry'
```

__str__(*self*)

```
    Returns the string representation of this 'NameTableEntry'.
```

getGUID(*self*)

```
    Returns the GUID of the 'NameTableEntry'.
```

getIP(*self*)

```
    Returns the dotted IP address of the 'NameTableEntry'.
```

getPort(*self*)

```
    Returns the TCP/IP port associated with this 'NameTableEntry'.
```

getRevision(*self*)

```
Returns the revision number with this 'NameTableEntry'.
```

incRevision(*self*)

```
Increments the revision number with this 'NameTableEntry'.
```

merge(*self*, *other*)

```
Merge this 'NameTableEntry' with another one.

Parameters:

    'other' -- 'NameTableEntry' to merge with
```

setGUID(*self*, *guid*)

```
Sets the GUID of the 'NameTableEntry'.

Parameters:

    'guid' -- new GUID (string/int) of the process represented
    by this 'NameTableEntry'
```

setIP(*self*, *ip*)

```
Sets the dotted IP address of the 'NameTableEntry'.

Parameters:

    'ip' -- a string containing a hostname, a dotted IP address,
    or '"*"' to represent the first public Internet adaptor's
    address.

Notes:

    The 'ip' parameter will be converted into a dotted IP address
    before it is stored.
```

setPort(*self*, *port*)

```
Sets the TCP/IP port associated with this 'NameTableEntry'.

Parameters:

    'port' -- new TCP/IP port number
```

NameTable Class

__init__(*self*)

```
    Initializes a 'NameTable' object.
```

__str__(*self*)

```
    Returns the string representation of the 'NameTable' object.
```

addEntry(*self*, *newEntry*)

```
Adds a 'NameTableEntry' to the 'NameTable'.

Parameters:

   'newEntry' -- 'NameTableEntry' to add to the 'NameTable'
```

enumGUIDs(*self*)

```
    Returns a list of all of the GUIDs represented by
    'NameTableEntry' objects in the 'NameTable'.
```

lookupEntry(*self*, *guid*)

```
Looks up a 'NameTableEntry' in the 'NameTable'. Throws
an exception when one is not found.

Parameters:

    'guid' -- string/int identifying the process whose
    'NameTableEntry' is being looked up

Returns:

    A 'NameTableEntry' corresponding to the guid.
```

merge(*self*, *other*)

```
Merge/add entries from another table into this one.

Parameters:

    'other' -- 'NameTable' object to merge 'NameTableEntries' from

Notes:

    This method will throw a 'MergeException' if you attempt to merge
    incompatible 'NameTable' objects.
```

resolveTaggedObj(*self*, *tObj*)

```
Resolves the destination IP, destination TCP/IP port,
source IP, and source TCP/IP port corresponding to a
'TaggedObject'.

Parameters:

    'tObj' -- 'TaggedObject' to resolve

Returns:

    A tuple containing (dstIP, dstPort, srcIP, srcPort)

Notes:

    Any of the resolvable elements of the tuple are filled in
    with a None object.
```

BIBLIOGRAPHY

[1] *Epidemic Algorithms for Replicated Database Maintenance* (1987), ACM Press.

[2] Bernstein, P. A. Middleware: a model for distributed system services. *Commun. ACM 39*, 2 (1996), 86–98.

[3] Chow, R., and Johnson, T. *Distributed Operationg Systems & Algorithms*. Addison Wesley Longman, Inc., 1997.

[4] Hewitt, C. Viewing control structures as patterns of passing messages. Technical Report 410, Massachusetts Institute of Technology - Artificial Intelligence Laboratory, December 1976.

[5] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[6] Krakowiak, S. http://middleware.objectweb.org/. What's Middleware.

[7] Lutz, M. *Programming Python, 3rd Edition*. O'Reilly, 2006.

[8] McMillin, B., and Arrowsmith, E. CCSP - a formal system for distributed program debugging. In *Proceedings of the Software for Multiprocessors and Supercomputers, Theory, Practice, Experience* (September 1994), pp. 260–269.

[9] Natarajan, B., Gokhale, A., Yajnik, S., and Schmidt, D. C. Doors: Towards high-performance fault tolerant corba. In *Proceedings of the 2 Distributed Applications and Objects1. Introduction Proceedings of the 2 Distributed Applications and Objects1. Introduction Proceedings of the 2nd Distributed Applications and Objects (DOA) conference* (September 2000).

[10] Object Management Group (OMG). *Portable Interceptor Specification*, OMG Document orbos edition ed. Framingham, MA, USA, December 1999.

[11] Object Management Group (OMG). *Common Object Request Broker Architecture: Core Specification*. Framingham, MA, USA, March 2004.

[12] O'Halloran, C. On requirements and security in a CCIS. In *CSFW* (1992), pp. 121–134.

[13] O'Halloran, C. *Category Theory and Information Flow Applied to Computer Security*. PhD thesis, Univeristy of Oxford, June 1993.

[14] Prechelt, L. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, March 2000.

[15] Roth, G. F. *Biologically Inspired Intrusion Detection in Distributed Systems*. Master's thesis, University Of Missouri - Rolla, 2003.

[16] Serban, C. *Run-Time Security Evaluation For Distributed Applications*. PhD thesis, University Of Missouri - Rolla, 1996.

[17] Siegel, J. *CORBA 3 Fundamentals and Programming*, 2 ed. John Wiley & Sons, Inc., 2000.

[18] Sun Microsystems, Inc. *Solaris Dynamic Tracing Guide*. 4150 Network Circle, Santa Clara CA 95054, January 2005.

VITA

Ian Jacob Baird was born to Barbara and Jason Baird at Grand Forks Air Force Base, North Dakota on January 5th, 1980. He was interested in computers and programming at an early age, and this passion led him to the Computer Science Department at the University of Missouri - Rolla, where he obtained his B.S. degree in Computer Science in December 2002. He received his M.S. degree in Computer Science in December 2007.