A FAULT-TOLERANT HOME-BASED NAMING SERVICE FOR MOBILE AGENTS

By

Cam Tolman

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Approved:

Carlos Varela
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2003
(For Graduation May 2003)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

For the research going into this thesis, I worked closely with members of two distributed computing research teams. The first team is referred to as the Worldwide Computing Research team. The individuals of this team provided a good amount of feedback on this research and as such deserve an expression of thanks. The second team is called the SALSA development team. The members of this team are also members of the first team, so I am thanking them twice. My second expression of thanks is in regards to the help they provided on the actual prototyping of the proposed naming service. The members of the SALSA development team put in a good amount of time and effort in making the SALSA distributed language more usable, and these efforts should not go unacknowledged. Keep up the good work!

I would also like to extend my thanks to my thesis advisor, Carlos Varela, who provided a great deal of assistance, guidance, and critical feedback on all the efforts made for this thesis.

And lastly, I should acknowledge my employer General Dynamics Advanced Information Systems in Pittsfield Massachusetts for the assistance they provided. Specifically, they covered all the cost incurred from attending RPI and they also allowed a few hours off on a weekly basis to attend classes and perform the research.

The courses and research that went into all my graduate studies have consumed a great deal of time and have lead to many restless nights. But, those days are coming to an end since I am now finished, and it is with the submission of this thesis. And, to be honest, it is this very completion that fills me with the most gratitude.

ABSTRACT

As a mobile software agent migrates to various host machines on a network, collaborating agents need to be able to locate the mobile agent for communication. A *naming service* is in charge of locating such an agent in a distributed system given its name. Three critical characteristics of a naming service are: *fault tolerance*, *scalability*, and *efficient name resolution*. Most naming services provide support for efficient name resolution but do not address fault tolerance or scalability issues. Conversely, distributed hash-table approaches to naming provide fault tolerance and scalability albeit at a sacrificed name resolution performance. We introduce a Fault-Tolerant Home-Based Naming Service (FHNS) that is robust and scalable yet enabling efficient name resolution.

An agent using FHNS has a unique name that encodes its *home base*. Home bases are connected in a peer-to-peer manner. The agent's run-time system is responsible for keeping the agent's home base informed of its network location. An agent obtains a reference to another agent by knowing the name of that agent and requesting any home base to resolve that agent's location. Redundancy exists between neighboring home bases to ensure the location of any given agent can still be resolved despite an agent's respective home base failing, thus eliminating single points of failure. In the event of a home base failure, the failover procedure occurs transparently to the agents. Furthermore, FHNS has optimal sequential fault tolerance behavior: if the home bases fail sequentially down to any single remaining home base, FHNS is still able to resolve the location of every agent in the distributed system.

Resolving the location of an agent with a distributed hash-table approach, such as Chord or SPRR, requires $O(log\ n)$ messages between home bases. In a worldwide computing system with potentially billions of nodes, such a lookup could still take up to 30 messages. Realizing that logarithmic lookups are not as efficient as needed for a naming service, FHNS provides name to location resolution in one round-trip request (two messages) in the general case. A logarithmic number of messages is only needed when a direct request to a home base fails.

FHNS is a novel contribution being made to the field of mobile agent computing. Fault-tolerant home-based naming can be incorporated into existing mobile agent platforms to eliminate single points of failure without sacrificing efficient name resolution.

CHAPTER 1
Locating Mobile Agents

A *naming service* maps a name for an entity to a set of labelled properties. The Domain Name System (DNS) [37], for instance, is a network naming service that maps *easy-to-remember* names to *hard-to-remember* network addresses. Similarly, name services can be put to use in distributed computing architectures to map the identity of a mobile software agent to its current location on the network. The Fault-Tolerant Home-Based Naming Service (FHNS), presented in this thesis, is one such naming service.

The motivation to develop a new naming service comes from considering the under-utilization of the Internet, which is predominately used for information retrieval, correspondence, commerce, and file sharing. The Internet, which can be described as a super-computer consisting of over a million processors, offers a vast amount of computing capability that has yet to be fully exercised. Substantial computations, such as data mining, financial forecasting, and complex simulations, can be divided and distributed across the Internet to be serviced by numerous machines concurrently. In order for these computations to be distributed, a framework first needs to be structured and deployed. The framework is required to be efficient, robust, scalable, and secure. The purpose of this research, however, is not to present a distributed computing framework in its entirety but to present a naming service that enables such a framework to meet a measure of the identified requirements.

The requirements for a naming service intended for a network like the Internet are equivalent to those placed on a distributed computing framework. The naming service must be:

- **Robust:** Any one failure cannot result in the entire service failing. The naming service must be resilient to node and link failures.
- **Efficient:** The messages associated with resolving the location of an agent should be many times less than the number of nodes on the network. Not every node can be queried when an agent is being located.
- **Scalable:** The service should allow nodes and agents to be located anywhere in the world and be able to support a huge number of these entities.

- **Secure:** The service must ensure data integrity and thus protect the names it is entrusted with from unauthorized modification or deletion.

And the names for the mobile agents must be:

- **Unique:** This is a requirement common to all naming services. In this case, the requirement for uniqueness is even more extensive in that the names for the mobile agents are to be globally unique.
- **Persistent:** The names must not change unless the naming service is notified.
- **Human Readable:** This is a requirement we place upon ourselves with the notion that human readable names are easier to remember and use. A human readable name can also indicate the purpose the agent serves, as would be the case of naming a personal-calendar agent: *JoeDoesCalendar*. Furthermore, people have the possibility to guess the agent name.

Mobile agents are best described as software processes that can migrate from one host to another while executing their assigned tasks. In between migrations, a mobile agent may need to interact with other agents to complete their jobs. Cabri, Leonardi, and Zambonelli [12] identify 4 kinds of coordination models for agents to employ in interacting with one another. The first model is based on *direct coordination* and involves the agents sending messages directly to other agents. Direct coordination requires that agents know the names of those agents with whom they wish to interact. The second coordination model is referred to as *meeting-oriented*. The meeting-oriented model does not require that agents know the names of other agents, instead an agent joins a known meeting point, where it can interact and synchronize with other agents. The third model is the *blackboard-based* model, where agents post and retrieve messages from a shared blackboard. One advantage of the blackboard-based model is that the agents do not need to exist at the same time. A sender can post and terminate before the receiver ever reads the message. The final model is the *Linda-like* model, which is based on the associative mechanisms of the Linda model of process creation and coordination [14]. As in the blackboard-based model, a space is used to store and retrieve messages, called tuples; however, unlike the blackboard-based model, the agents retrieve the messages in an associative way.

The mobile agents being considered in this thesis communicate directly with one another and consequently direct coordination mechanisms are required. Say for instance,

a protein folding application decomposes the computations associated with protein folding into sub-computations to be processed by mobile software agents. These software agents are dispersed throughout the Internet to perform their distributed tasks. After performing a number of its computations, software agent GYPSY needs Software Agent NOMAD's results in order to continue with its computations. How does GYPSY locate the whereabouts of NOMAD given that it could be at any node on the Internet? What follows are possible strategies to address the problem of locating an agent for the purpose of communicating directly with that agent [45, 50, 24].

1.1 Broadcast

One strategy would be for GYPSY to broadcast a request for NOMAD's results or current location. Software agent NOMAD would receive the requested broadcast and would reply accordingly. Estimates on the size of the Internet are in the tens of millions [62]. This solution is thus too expensive since every node on the Internet would receive all such requests and consequently the scalability and efficiency requirements would be breached.

A variation of the broadcast would be to only query a portion of the nodes on the network as is done by issuing a *multicast*. GYPSY would be required to keep a list of all possible locations where NOMAD could be. However, we assume agents are free to roam the entire Internet and are not constrained to only move within a pre-defined subset of nodes. Given that we do not place constraints on agent mobility, we cannot identify the subset of nodes to query.

1.2 Search

Another solution would be for GYPSY to search for NOMAD. But how does GYPSY conduct an efficient search for NOMAD? In the worst case, the search method would be as expensive as a broadcast in that GYPSY contacts every node in its search for NOMAD. In the best case GYPSY only has to query a small amount of nodes by minimizing the search space. But how would this be done? Agents are not required to keep an itinerary for themselves and so, as it was with a multicast, the subset of nodes could never be fully identified. For the sake of argument, if we knew in advance an agent's route of travel, then an efficient searching algorithm would not be impractical. However, we assume the mobile agents only know of each other by name and know nothing of each other's routes

of travel. So, given this assumption, searching is not a feasible solution.

1.3 Tracking

Tracking is performed with the use of forwarding pointers that are distributed along the path of the traversing mobile agent. Each time the agent changes its location a pointer to the new location is deposited at the old location. So GYPSY locates NOMAD by following the "bread crumbs" left by NOMAD. The problems that must be overcome to use this service are numerous. First how does GYPSY determine the initial location of the path to follow? An implicit requirement is placed on agents to know the first destination of the agent being sought. Assuming that GYPSY does locate the path NOMAD has left, a race condition could ensue between GYPSY and NOMAD. Also this naming service is prone to failures anywhere along the chain of pointers and it does not allow these forwarding pointers to be deleted in its simplest form.

Forwarding pointers are useful in other applications such as message forwarding and distributed garbage collection [40] but in their simplest form, forwarding pointers are not well suited for providing a naming service.

1.4 Centralized Dedicated Naming Service

Unlike the previously proposed strategy, the *central service* is a more traditional naming service in that it follows the convention of a server and a request-issuing client. In this naming service there is a single source containing a listing of all agent names that map to the present location (i.e. host) of the agent respectively. So when GYPSY needs the results from NOMAD, it would contact the naming service and the location of NOMAD would be provided. Locations could be resolved with a single message loop: a request and a reply. This is the best solution in terms of network communication. However, the problem with this solution is that there is now a central point of failure. The central naming service exists at a single location, which if that location fails then the entire service fails. As already specified, the framework needs to be robust. Replication could be used to achieve fault tolerance. But then this centralized service would have to handle all requests, potentially on the order of "tens of millions" at a time. A bottleneck would be the result and consequently the overall performance of the framework would be severely degraded. Applying this solution would prevent us from meeting the scalability requirement.

1.5 Distributed Dedicated Naming Service

Now if the central naming service is distributed, the bottleneck problem is eliminated. A naming service that is "distributed and centralized" may appear to be an oxymoron but it simply means that the network address mappings are not all at the same location. Each software agent has a *home base* that keeps track of where that particular software agent is at all times. So GYPSY would contact NOMAD's home base, which would then provide GYPSY with the current location of NOMAD. As with the central naming service there still exists a single point of failure but it is on a per agent/agents basis as opposed to a single point of failure for the entire system. Replication can be employed to address this single point of failure drawback.

Also with the home base scheme, the original question of how to find mobile agents, becomes a question of how to find the home base for a given agent? With a central naming service, all agents go to a single source for the information they seek and all agents can be provided the location of this single name server. There are numerous name servers with the home base naming service and the agents must either know the specific name server that stores the information it needs or the agents will need to know how to find a particular name server. Since the home bases, which function as name servers, do not ever change location, they will be easier to locate than transient agents.

Obviously, the Fault-Tolerant Home-Based Naming Service follows this very scheme. Jumping ahead, it will be shown that this proposed naming service includes two ways of locating the home base for a mobile agent: a *location-dependent* mechanism, which makes locating trivial, and a *location-independent* mechanism.

Structure of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 presents the model of the Fault-Tolerant Home-Based Naming Service including an assessment of its main desired features. Chapter 3 presents an FHNS implementation that was tested. The results of these tests are presented and evaluated in Chapter 4. Chapter 5 presents related work in the area of naming services in distributed computing. And lastly, Chapter 6 and Chapter 7 presents some concluding remarks in regards to future work and the contributions of this research.

CHAPTER 2
Fault-Tolerant Home-Based Naming Service (FHNS)

2.1 Model

The Fault-Tolerant Home-Based Naming Service conforms to the home base naming service construct and as such each mobile agent has a home base that keeps track of that agents location. The mobile agents can either specify their home base directly or can opt to have a home base assigned. The naming service recognizes and is capable of resolving two types of names: location-dependent names or location-independent names. Due to this versatility, the service can effectively shift from a location-dependent service to a location-independent service in the event of a node failure (i.e. home base failure).

FHNS is succinctly comprised of the following:

1. **Unique Names:** Mobile agents have unique names that are premised upon Uniform Resource Identifiers (URI) [9]. The URI can take the form of a URL [8], which is location-dependent and is how the agent specifies its home base, or the URI can take the form of a URN [36], which is location-independent and requires a home base to be assigned[1].

2. **Home Bases:** Nodes on the network that provide the naming service for the mobile agents.

3. **Mapping Records:** A record kept for a mobile agent that is kept current with the agent's location on the network. The record is stored at the agent's respective home base.

Table 2.1 details the API for the Fault-Tolerant Home-Based Naming Service, which are the specific home base services.

2.2 Names

The location-dependent naming is attractive for the obvious reason that the location of the home base is given explicitly. Consequently, the agents themselves can be

[1]The names adhere to the rules specified in the references with the one modification being made to the names following the URN format.

Function	Description
`put`(name, location)	Binds a name to an initial location.
`get`(name)	Returns the location for the given name.
`update`(name, location)	Updates the location of the agent with the specified name.
`delete`(name)	Remove the location record for the agent with the specified name.

Table 2.1: Fundamental API for FHNS

found quickly and efficiently simply by querying the respective home base. The location-dependent names of an agent take the form of a URL where the general syntax is:[2]

<location-dependent name> ::= <scheme>:<scheme-specific-part>.

A URL contains the name of the *scheme* being used, which is an existing or experimental protocol. The scheme is followed by a colon and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme. For Internet schemes the scheme specific part is written as follows:

//<user>:<password>@<host>:<port>/<url path>.

The optional elements are "<user>:<password>@", ":<password>" and ":<port>". The scheme specific data begins with a double slash "//" to indicate that it complies with the common Internet scheme syntax. The *host* field corresponds to the domain name of the node, or home base for the purposes of FHNS. The *url path* is not an optional element for FHNS and corresponds to the relative name of the agent. The names for an agent are referred to as either relative or absolute. The absolute name refers to the name in its entirety. The relative portion of the name refers to the name given to the agent that distinguishes it from other agents using the same host as a home base.

Location-independent naming is appealing because it allows the home base to change for a given agent. With location-independent naming, the names of an agent can take the form of a URN. The URN syntax is:[3]

"urn:" <NID> ":" <NSS>.

The *NID* is the *Namespace Identifier*, and *NSS* is the *Namespace Specific String*. A ":" delineates the three elements. The "urn" descriptor string is specified to be a required

[2]Refer to [8] for further details on the specific syntax of a URL.
[3]Refer to [36] for further details on the specific syntax of a URN.

element; however, for the purposes of FHNS this string will be replaced by another string corresponding to an existing or experimental protocol. This modification allows for the two types of names to share a common element. The FHNS modified URN will be written as

<location-independent name> ::= <scheme> ":" <NID> ":" <NSS>.

A URN differs from a URL in that its primary purpose is persistent labelling of a resource with an identifier as opposed to serving as a locater. The location of agents are then resolved by using *consistent hashing* [30] and a lookup service such as the Chord [49, 19] peer-to-peer lookup service. By using a peer-to-peer lookup service, the home bases are required to keep a listing of other home bases so that when a *lookup* is required a home base will be able to query other home bases for the information it needs, in this case the location of an agent. A lookup is the process of a home base querying other home bases and could be performed when servicing any of the naming service requests. A lookup is not synonymous with the `get` service request.

2.3 Architecture

FHNS can be described as a layered architecture with DNS serving as an underlying naming service. DNS is utilized by FHNS when resolving the domain name of a home base to an IP address.

Names in the Domain Name System are apart of a naming hierarchy, with the most significant part on the right. The left-most portion of the name corresponds to the name given to the computer. Clients query local DNS servers for IP addresses. The Local server starts with the root name server and recursively queries DNS servers until it finds a server that has the answer. In the case of a client request regarding *www.yahoo.com*, the local server queries a root DNS server then a *com* DNS server, and then finally the *yahoo.com* server for the web server IP address.

Fault tolerance is defined as the ability of a system to respond gracefully to an unexpected hardware or software failure. Fault tolerance for DNS is achieved with the use of a designated primary server and a secondary or "back-up" server (or servers). If the primary server fails, requests are directed to the secondary server. With this approach, however, the notion of *failover* is always provided for the primary but not the secondary. If the secondary server fails or the last "backup" server fails, there exists a fault in the

system. FHNS achieves fault tolerance with redundancy between home bases and provides for an iterative failover capability. Each home base functions as both a primary and secondary server. And each home base has a designated successor that is capable of servicing requests on behalf of that home base in the event of a failure. Essentially, FHNS is capable of recovering from all home bases, except one, failing.

Returning our attention back to DNS, when an organization wants to participate in the Domain Name System, the organization must apply for a name under one of the top-level domains (i.e. com, edu, gov, etc.). If the name is already in use, the organization will not be allowed to use the name and thereby uniqueness is ensured among all the top-level domain names. If the name is not already in use and the organization is allowed to use the name. They then assume the responsibility of ensuring that names are unique with respect to their domain. Names are kept unique in FHNS in a similar manner. In the case of location-dependent naming the first part of the name, that being the host, must be unique since it is a name in the Domain Name System. The name of the agent must then be unique with respect to the home base (i.e. host). The home bases reject a put request if the name requesting to be used is already in use—thereby ensuring uniqueness. Ensuring unique naming with the location-independent names requires that both the namespace identifier and the namespace specific string be unique. Again the home bases can reject a put request if the namespace specific string is not unique relative to the namespace identifier. A single global namespace identifier will be used, which is thus trivially unique (multiple namespace identifiers are discussed in Chapter 6).

The namespace identifier corresponds to a logical ring with distinct points along the ring called *identifiers*. A single home base is assigned to one of the identifiers by way of consistent hashing. Consistent hashing is a distribution scheme that pertains to using a hash function to distribute strings to a number (i.e. identifier) in the range [0,..., M], where M is determined by the selected hashing function. Home Bases are assigned to an identifier by hashing the domain name or the IP address. The URN of an agent is also hashed and mapped to an identifier. The agent is then assigned a home base by identifying the home base that has been mapped to a number greater than or equal to the number the agent has been mapped to (0 is the first number greater than M, when making assignments). All home bases belong to this single logical ring and each home base keeps a routing table that lists other identifiers and the home bases associated with

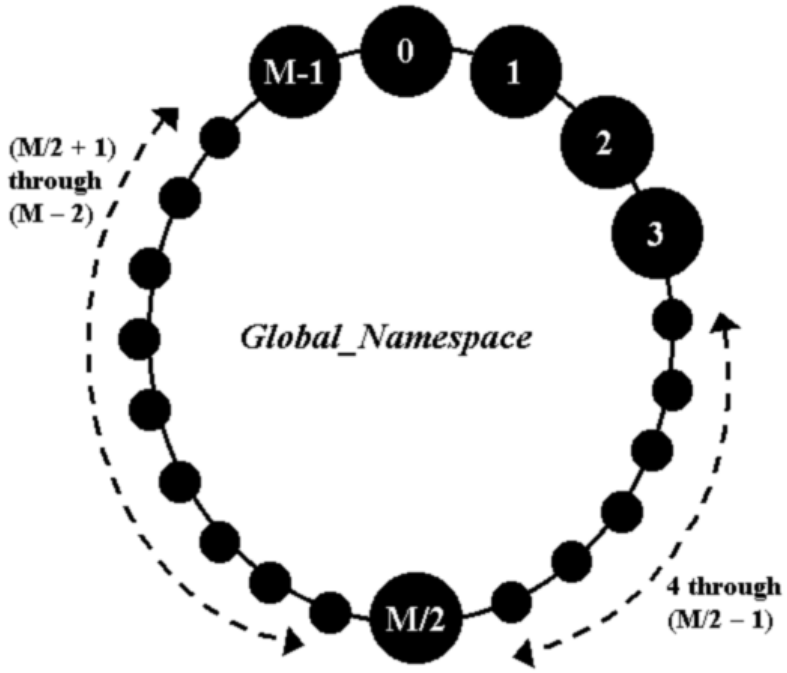

Figure 2.1: FHNS namespace depicted as a logical ring with M identifiers.

those identifiers. The routing table contains $log\ M$ entries[4]. Each home base has its mapping records replicated at a number of home bases with identifiers greater than and nearest to the identifier of the respective home base. The logical ring is not static, new home bases can join and the mapping records will be transferred accordingly. Conversely, home bases can leave the logical ring and one of the replicating home bases will assume responsibility for the departed home base's mapping records. However, the hosts that will serve as home bases should be chosen judiciously since it would not be desirable to have a home base that is only powered on for short durations, as would be the case with mobile devices. Also, home bases are not restricted to being dedicated name servers only. Home bases can be like other hosts in providing a platform for mobile agents to execute their assigned tasks. Figure 2.1 depicts the FHNS namespace as a logical ring consisting of M identifiers.

[4]Unless stated otherwise, all logarithms in this thesis are of base 2.

2.4 Protocol

As an example of how the naming service functions, we now present a description of agents using the service. We begin with a 3-bit namespace, called $drifters$, and we will use *SHA-1 % 8* [59] as our hashing function. We identify 3 hosts to be our home bases and assign them each an identifier.

NAME	IDENTIFIER
rome:3030	7
hongkong:4040	0
newyork:5050	5

Table 2.2: Example: Home Bases and the Assigned Identifiers

In practice we would use a hashing function bigger than 3-bits to avoid collisions and the identifiers would not be assigned but determined by hashing the domain name. Next we introduce 4 mobile agents, half will use location-dependent names and the other half of agents will use location-independent names. Our experimental naming service scheme will be called $fhns$. Figure 2.2 illustrates these assignments.

RELATIVE NAME	ABSOLUTE NAME	IDENTIFIER
MIGRANT	fhns://rome:3030/MIGRANT	6
TUMBLEWEED	fhns://hongkong:4040/TUMBLEWEED	0
GYPSY	fhns:drifters:GYPSY	4
NOMAD	fhns:drifters:NOMAD	2

Table 2.3: Example: The Agents' Names and the Determined Identifiers

The agents would need to be aware of at least one home base so that they could issue a request to that home base to perform a put(<agent absolute name>, <agent location>). Agents TUMBLEWEED and MIGRANT could issue the request to the host specified in their names, which would then become their corresponding home base. GYPSY and NOMAD could issue their requests to any of the home bases. The home base servicing the request first hashes the names of these agents, GYPSY and NOMAD, to determine the proper home bases and then forwards the request to those identified home bases. home bases are able to forward requests by determining from their routing tables the host nearest the identifier they are seeking.

If GYPSY wanted to communicate with MIGRANT it could request that the *rome* host machine perform a `get`(*fhns://rome:3030/MIGRANT*) operation. The location of

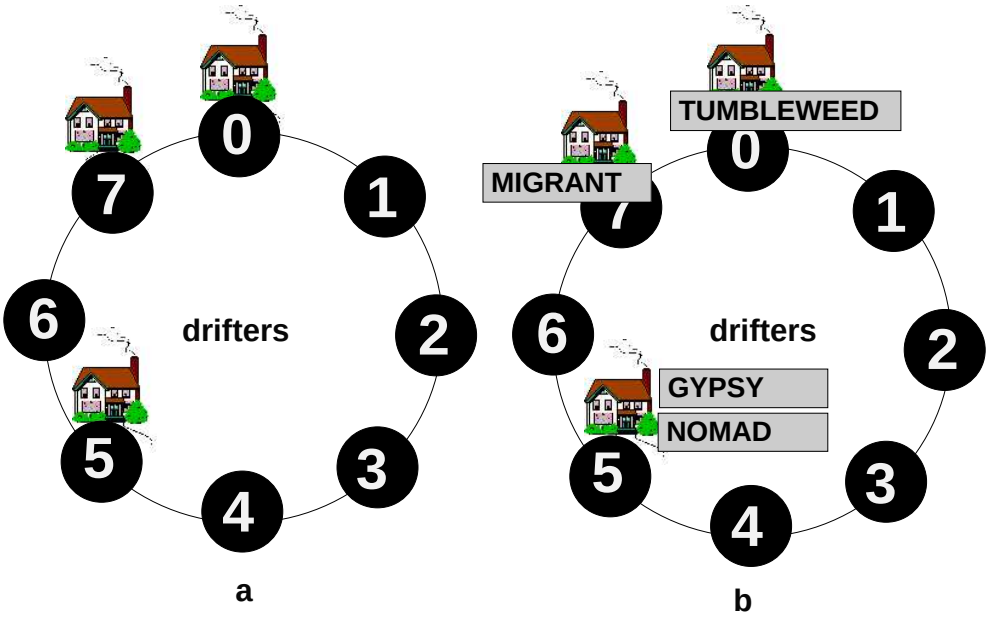

Figure 2.2: (a) Populated 3-bit identifier space (b) home bases with mapping records

MIGRANT would be provided with a single message loop (i.e. request and reply), which is why we say that the location-dependent naming aspect of this naming service is very efficient in resolving actor locations. But say GYPSY wanted to communicate with NOMAD. GYPSY would direct a request to any of the home bases to perform a `get`(*fhns:drifters:NOMAD*) operation. The home base, say *hongkong*, would first hash the string "NOMAD", which is the agent's relative name, according to the *SHA % 8* hashing function associated with the *drifters* namespace. The relative name of NOMAD hashes to identifier 2. *Hongkong* would look into its routing table and see that *newyork* maps to identifier 5, which is the node greater-than and closest to identifier 2. The `get`(*fhns:drifters:NOMAD*) request would be handled by *newyork* and the results would be forwarded to *hongkong* and then provided to GYPSY. Here it took more than a single message loop to find the location of NOMAD. Location-independent naming does not give as good results in resolving the location of agents since it requires that the home base for an agent first be discovered. Finding the location of a home base could take $O(log\ n)$ messages when using the Chord lookup strategy, where n is the number of nodes in the ring (i.e. home bases). When a home base is requested to determine the location of an agent, it first checks the mapping records in its possession. If a mapping record is found,

it returns with the location of the agent. If a mapping record is not found, then the home base forwards the request to the home base that is closest to the identifier to which the name of the agent is mapped. The request is forwarded until it is serviced by the authorized home base. To minimize the number of hops between nodes, the searching home base can cache the location of an agent's home base so as to avoid having to perform another multi-message lookup for subsequent locate requests for the same agent.

Some time has passed and NOMAD decides to move from a host machine called *host1* to *host2*. It first requests a home base to perform an `update`(*fhns:drifters:NOMAD*, *host2*) operation and then NOMAD moves to the new host. The agents are required to provide a mechanism to handle messages in transit that were sent to the old location of the agent. One way of handling this would be for the agent to leave behind a forwarding pointer at the old location directing messages to the new location. The forwarding pointer could be deleted by expiration or from a request made by the originating agent.

When the agents are done performing their tasks, they are able to remove themselves from the system by issuing a `delete`(<agent absolute name>) request.

When a home base crashes and is removed from the network, the agents using that home base still need to have some way of being located. The agents are still on the network but the locating mechanism has been disrupted, and not addressing this contingency would render these agents *unreachable*. Names that are location-independent lend themselves nicely to fault tolerance since they are not bound to any home base explicitly. Using Chord, we start with a logical ring and place home bases at distinct points along the ring based upon their corresponding identifiers. Moving in a clockwise fashion, the subsequent home bases keep copies of the mapping records belonging to the previous home bases. So in the event that a home base fails, the subsequent home base will be able to service all the requests that would have been handled by the now departed home base. Agents do not become unreachable when their home base fails. The agents using location-dependent names can still be found because the host portion of their names corresponds to a home base and thus maps to an identifier. The subsequent identifier can easily be determined and the replicating home base can thus be found. This does introduce a incongruity in that the location-dependent name does not accurately give the location of the agents home base, but there is no way to solve this problem since we wish names to be persistent even in the event of a failure. This incongruity can be avoided by using location-independent names.

2.5 Assessment

Desirable features are realized when allowing both location-dependent naming and location-independent naming. These desirable features include fault tolerance, efficient location resolution, and scalability. Conceptually, FHNS meets the requirements specified for a large-scale naming service. The responsibility in meeting these requirements falls upon the home bases.

2.5.1 Robustness

A fault tolerant infrastructure is erected when using consistent hashing and a lookup service such as Chord. All home bases belong to this infrastructure. Although this infrastructure is primarily used for locating agents in the location-independent case, it can be used to achieve fault tolerance in both the location-dependent naming and location-independent naming cases. If a home base failure occurs, the subsequent home bases will be able to provide redundancy for the mapping records belonging to that failed home base. With location-independence, the lookup will advance around the circle until a mapping record is found. In the location-dependent case, we know the home base that is unresponsive since we know the home base of the agent. So all that needs to be done is to determine the home base that is subsequent to the failed home base in the logical ring and have all requests serviced by the subsequent home base.

2.5.2 Efficiency

It is clear that the best performance is obtained with location-dependent naming since it requires less messaging and no searching for the respective home base. In fact the performance with location-dependent naming is as good as the performance of the centralized dedicated naming service strategy. The performance is not as good with location-independent naming but FHNS is amenable to optimizations such as caching. Caching can be employed by the home bases to emulate the location-dependent naming performance when location-independent naming is being used. An agent makes a request for a naming service to be performed by a specific home base and if a lookup is required, that home base can cache the results so that a search is not needed when asked to perform another naming service for that same agent.

If caching is used, resource efficiency becomes an issue. The number of agents to be cached should be capped so that resources are used efficiently. There are a number of considerations for choosing the optimum cache size, which are beyond the scope of this

thesis. Essentially, the number of agent locations that are cached should be large enough so as to minimize the cache miss rate and yield better performance.

2.5.3 Scalability

FHNS is scalable in terms of messaging when location-dependent naming is used. It is likewise scalable with location-independent naming by using a scalable lookup service, such as Chord. As previously stated, Chord resolves all lookups via $O(log\ n)$ messages to other nodes. Furthermore, Chord is able to insert new Home Bases or remove failed Home Bases from the appropriate routing tables of existing Home Bases with $O(log^2\ n)$.

Load balancing is defined as distributing processing, storage, and communications activity evenly across servers so that no single server is overwhelmed. Consistent hashing has the property of distributing keys among the identifiers in a relatively even manner and as a result no single server is overwhelmed in regards to storing the keys. Consistent hashing is only used with the location-independent names and so this load balancing feature can be comprised with location-dependent names. To address this problem, home bases are permitted to reject a `put` request containing a location-dependent name if that home base is overloaded compared to its immediate neighbors. Further discussion is given in Chapter 4 on how to use Chord to better distribute the keys.

A *federated* naming service is one that delegates control to various name servers to eliminate a centralized bottleneck or single point of failure. Federation is of critical importance in meeting the scalable and robust requirements. For instance, DNS is a federated naming service that composes a global namespace for the Internet through a hierarchy of lower level namespaces, with each namespace delegating control to the next level down. FHNS is also a federated naming service but on a peer-to-peer basis as opposed to the hierarchy structure of DNS.

2.5.4 Security

Security has not been entirely incorporated into FHNS; however, solutions have been identified to address the security requirement. Data integrity can be ensured by using a public-key encryption scheme as has been proposed for use by other naming services such as DNS [10, 16, 18]. FHNS can be extended to incorporate these proposed security extensions.

Using a public-key encryption scheme, malicious agents can be prevented from issuing an `update` or `delete` request on behalf of other agents. Likewise, a public-key

encryption scheme could be used to protect agents from fraudulent home bases that provide erroneous data.

Presently, the home bases are capable of providing a level of security by denying `put` requests if the name an agent wishes to go by is already in use. In so doing, the home bases ensure that names are unique. And here again the public-key encryption scheme could be applied to allow the rebinding of a name already in use.

CHAPTER 3
Implementation

An instance of the Fault-Tolerant Home-Based Naming Service has been prototyped using Chord as the lookup service. The prototype was then analyzed using the World-Wide Computer (WWC) [56] as a test bed. What follows are brief descriptions of the WWC and Chord. The architecture of the prototype is then described.

3.1 Context: World-Wide Computer (WWC)

In Chapter 1, we discussed frameworks that enable distributed computations to be performed. One such framework has been constructed with the intent of being used on the Internet. This framework recognizes the Internet's vast computing capability with its very name: the World-Wide Computer. The WWC enables tasks to be distributed among participating computers. The present naming scheme for this framework uses a home-based strategy and location-dependent names, which have an identical format to the location-dependent names for FHNS. The name servers do not include any fault tolerant features. So when a name server fails the agents that are bound to that name server become unreachable to the other collaborating agents. FHNS is used to eliminate this single point of failure.

The Actor model for concurrent computation in distributed systems [1] defines precisely a type of mobile agent known as an *actor*. An actor provides a unit of encapsulation for both state and a thread of control manipulating that state. Communication between actors is purely based on asynchronous message passing. Actors are reactive entities that respond to messages. Referring to Figure 3.1, a message triggers an actor to take one of three courses of action, the actor can: (1) modify its own state, (2) send messages to known actors, or (3) create new actors. Along with a state and single thread of control, each actor has a unique *mail address* to where messages can be sent. These messages are buffered in the actor's *mailbox* until the actor is ready to service them. There are no restrictions on the ordering of messages, and as such there is no guarantee that the actor processes messages in the order that they were sent. The only guarantee is that the actor will eventually process any message sent to it. In the WWC actors are extended with the capability of migrating from one node to another and are called *universal actors*.

SALSA is an actor programming language based on Java [57]. SALSA stands for

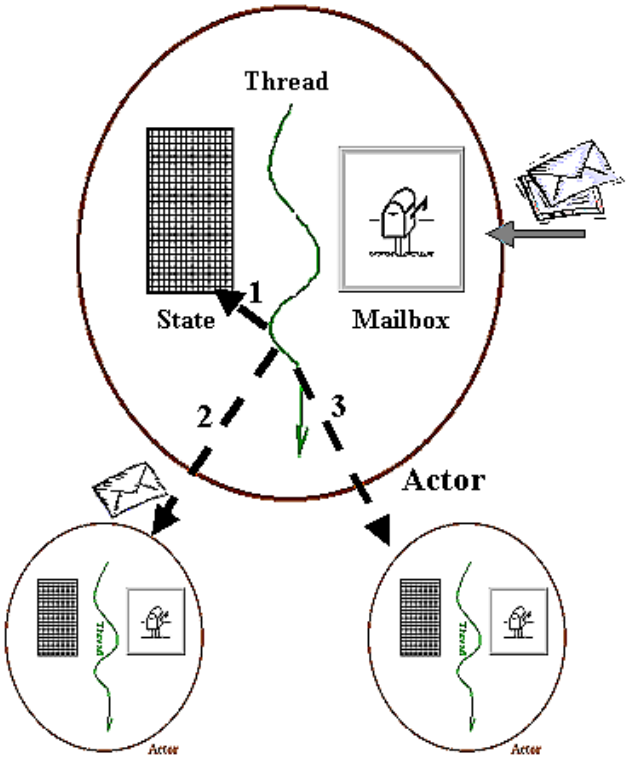

Figure 3.1: Actor Model for Concurrent Computation.

Simple Actor Language, System, and Architecture. The language simplifies programming dynamically reconfigurable applications for the WWC by providing universal names, active objects, and migration. Programs written in SALSA are compiled into Java source files [29]. The Java source files are then compiled to produce the bytecode recognized by Java Virtual Machines [35]. The SALSA language preserves many of Java's useful object-oriented concepts such as encapsulation, inheritance, and polymorphism. In SALSA, programs are grouped together in modules versus Java packages. And whereas a Java program consists of classes, objects, and methods, the SALSA program consists of behaviors, actors, and messages (the state of an actor is composed of private Java objects). A key difference between a method and a message is that objects block when a method is invoked, but actors do not block when they send a message due to the asynchronous nature of communication. An implementation of FHNS has been incorporated into the SALSA language so it can be used in the WWC.

The World-Wide Computer has been introduced to make it possible to pool the computing ability of devices that can communicate over a wide-area network such as the

Internet. The WWC consists of a set of virtual machines, or *theaters*, hosting universal actors. The universal actors are reachable throughout the Internet by their globally unique Universal Actor Names (UAN). The UAN identifiers persist over the lifetime of an actor and are used to obtain the actor's current theater. The Universal Actor Locator (UAL) represents the location of this theater. Universal actors extend actors in that they are able:

- to bind to a unique global name and to an initial theater,
- to obtain references to their peers from their names,
- to communicate through message passing in a location-independent manner, and
- to migrate from one theater to another.

In summation, the WWC is comprised of universal actors, theaters, and dedicated name servers.

An example of a UAN is

uan://wwc.publisher.com:3030/poemAgent.

The relative name of this actor is *poemAgent*, *wwc.publisher.com* is the name server or home base, and *uan* is the naming service protocol. The UAN has an identical format to the location-dependent names of FHNS. The naming server keeps tabs on where the agent is located be receiving information in the form of UALs. In this example, say the actor's UAL is

rmsp://wwc.editor.com:4040/poemAgent.

The naming server knows from this UAL that the poemAgent is located at the *wwc.editor.com* theater. The *rmsp* refers to a Remote Message Sending Protocol that enables sending messages to remote actors. The format for UANs has been augmented so as to support location-independent names for FHNS.

3.2 Context: Chord

Chord is a distributed lookup protocol that addresses the problem of efficiently locating data in peer-to-peer networks. The Chord protocol supports just one operation: given a key, it maps the key onto a node. The key can be associated with a particular data

item and together both the key and data are stored at the node to which the key maps. The data is then located by a *lookup(key)* algorithm that yields the value associated with that key, which could be the IP address for a given node on the Internet.

The consistent hash function assigns each node and key (such as the name of an agent) in the system an m-bit identifier. The node identifiers can be established by hashing the IP address or domain name. Likewise the identifiers for the keys, which are strings, are established by hashing the keys themselves. As with any hash function, collisions can occur. Collisions between keys are unimportant since keys are allowed to map to the same identifier and it is the keys themselves that must be unique. Collisions between nodes must be avoided in order for Chord to work correctly. Chord does not prevent collisions. FHNS can prevent a node collision at the point a node attempts to join the system by not allowing the node to join if it hashes to an identifier already in use. This solution would not work if two nodes join simultaneously and they both hash to the same identifier. But a simultaneous join collision is an unlikely occurrence so it will not be accounted for by FHNS.

Once the keys and nodes have been mapped to identifiers, the keys are then assigned to nodes. Any given key is stored at the first node whose identifier is equal to or greater to the key's identifier, where the last identifier in the identifier space is less-than the first identifier to effectively give a logical ring or an *identifier circle*.

Each node maintains a routing table with m entries, where m is the number of bits in the binary representation of the key or node identifiers. The routing table is called the node's *finger table* and the i^{th} entry in this table is given by $n + 2^{i-1}$, where n is the identifier of the node and $1 \leq i \leq m$. The node storing the identifier given by the i^{th} entry is listed in the finger table as the node to forward requests to when the $n + 2^{i-1}$ identifier is sought. Also listed in a node's finger table is the node previous to it in the identifier circle and is referred to as the *predecessor*.

If a node gets a lookup request for an identifier not listed in its finger table, it forwards the request to the entry in its finger table that lists an identifier closest to, but less than or equal to, the requested identifier.

In a dynamic network, nodes can join and leave at anytime. When a node joins the ring it has to request that an existing node provide the location of its nearest neighbors in the identifier space. Once the joining node receives its finger table information, it then requests that the keys it is responsible for be transferred. When a node leaves the network

gracefully, it transfers its keys and has its neighbors update their finger tables accordingly. Nodes leave and join the Chord identifier circle at a cost, with high probability, of $O(log^2\ n)$ messages.

Each node's nearest neighbor or neighbors keeps copies of the keys belonging to that node. So when a node fails, the nearest neighbor of that node can service the *lookup* operation directed to the failed node. Chord does not specify the number of neighbors that should provide redundancy but this number does correspond to the amount of fault tolerance attainable. If ten neighbors provide redundancy, then the system can conceptually recover without losing any of the keys in the event of ten simultaneous node failures.

The nodes perform stabilizing procedures in which they determine if their predecessor and immediate successor information is correct. By performing stabilization, the immediate neighbors of a failed node will see that it is not responding and will take the appropriate measures to recover from the failure by updating their finger tables. Chord does not specify how often to perform stabilization; however, the Chord test-case implementation performed stabilization at an average rate of 0.5 invocations per minute.

3.3 FHNS Prototype Developed for the WWC

The WWC already has a naming service in place that uses location-dependent names. This naming service is first extended to use location-independent names. A UAN can take the traditional form, specifically

uan://<home base host>:<port>/<actor relative name>.

A UAN can also take the location-independent form,

uan:WWC:<actor relative name>.

The namespace identifier, for this prototype, is called *WWC* and the corresponding hashing function to be used will be *SHA-1* [59], which hashes the given inputs to a 160-bit identifier. The namespace will encompass all identifiers greater than or equal to 0 and less than

$$10000000000000000000000000000000000000000_{16}\ or$$

$$1461501637330902918203684832716283019655932542976_{10}$$

A unique type of theater has been developed to serve as the home bases for FHNS and are called *home base theaters*. The naming service has been incorporated into these theaters instead of using dedicated naming servers. This is done to better ensure fault-tolerance. The alternative would be to have the actors keep a listing of home bases that they could query. A *Denial of Service* (DoS) could result if the home bases that an actor had a listing for had all failed. Having the actors direct all requests, with the exception of the `put` request, to the theater where they are presently "performing" ensures that the naming service is always available to the actors.

The test case consists of universal actors and home base theaters. These home base theaters host the universal actors and also serve as the name servers. Figure 3.2 depicts the structure of the home base theaters in the WWC. The figure also illustrates the other services provided by these theaters. In keeping with the "Broadway" manner of speaking, the home base theaters have a single *stage manager* that listens on a particular port for requests. Upon receiving a request, the stage manager delegates responsibility to servicing the request to a *stage hand* that is chosen from a thread pool. As illustrated in Figure 3.2 a home base theater can service a naming request using the Universal Actor Naming Protocol (UANP), a message directed to a hosted actor using the Remote Message Sending Protocol (RMSP), a hosting request also using (RMSP), or a coordination request using the Home Base Coordination Protocol (HBCP).

The methods belonging to the Universal Actor Naming Protocol are identical to the methods in the API for the Fault-Tolerant Home-Based Naming Service. The Fault-Tolerant Home-Based Naming Service is thus capable of servicing all UANP requests. One modification to UANP that was needed was to change it from being a strictly text-based protocol to being able to use Java objects. This was done to facilitate communication in that all requests could come into the home base over the same port.

When a message is received via RMSP, the stage hand servicing the request obtains from the *registry* a reference to the universal actor to whom the message is intended and places the message in the actor's mailbox.

A migrating actor comes into the home base theater through the same port as the other types of communications. The actor is registered with the home base and then is left to act upon the messages in its mailbox.

The Home Base Coordination Protocol provides the methods necessary for home bases to transfer needed information to participate in the namespace (logical ring). Ta-

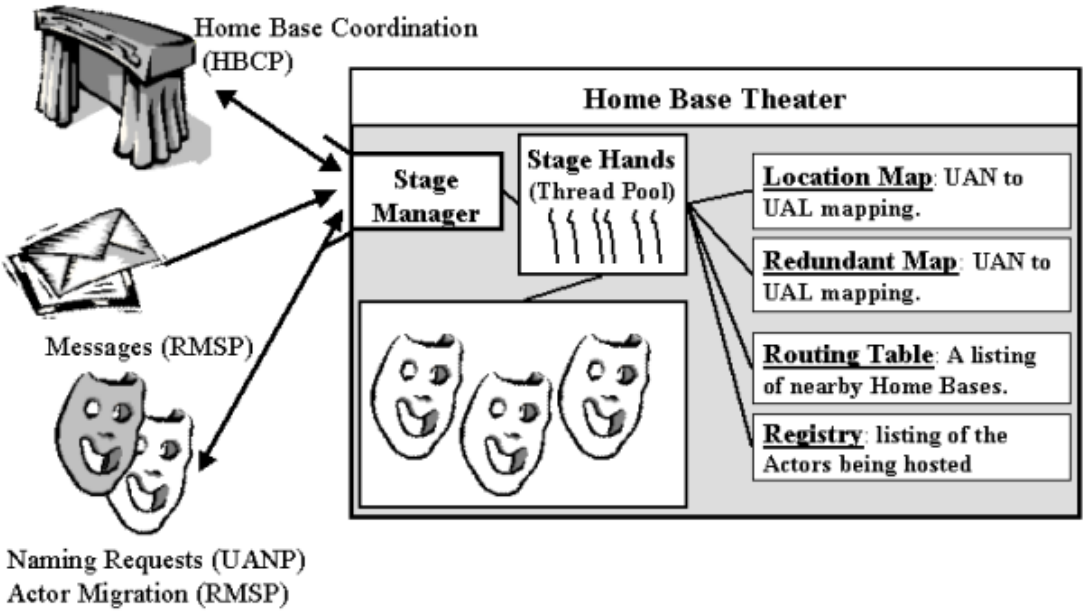

Figure 3.2: A Home Base Theater in the WWC.

ble 3.3 details the different HBCP requests and corresponding services.

The system is initialized by a single home base theater being started. The domain name of the machine the home base theater is running on along with the port number is hashed according to the *WWC* namespace hashing algorithm and an identifier is assigned. Notice that the domain names are being hashed and not the IP addresses. This is not to say that the IP address cannot be used, they in fact can. However, for any given node either the domain name needs to be used in all cases or the IP address since the IP address and domain name will hash to different identifiers.

Other home base theaters can then join the system by contacting a home base theater that already belongs to the system and retrieving the necessary routing table information. A joining home base theater, or node as it is referred in the proceeding explanations, first determines its identifier. It then issues a `nodejoin` request to an existing node. The servicing node returns with the name of the joining nodes immediate successor. The joining node then issues an `initialize` request to its immediate successor. The successor then provides the name of the joining node's predecessor and the nodes to list in its routing table and also the nodes that need to be informed of this node joining. The joining node then requests its successor provide the keys that it is responsible for with a `getkeys` request.

Request and Service	Description
NODEJOIN	Request made by a joining node where the servicing node returns with the requesting node's immediate successor
INITIALIZE	A joining node obtains its predecessor and routing entries from its successor
UPDATECOVERAGE	Joining or leaving node is inserted or removed and requests its successor and predecessor to update their Namespace coverage
GETKEYS	A joining node is provided the keys it is responsible for by its successor
REPLICATE	A node requests its successor to replicate the specified actor mapping
UPDATETABLE	A node has either joined or left (failed) and other nodes are informed to update their tables
GETNODE	Gets the node covering the region where the given Namespace identifier belongs
STABILIZE	A node periodically confirms the presence of its successor and predecessor

Table 3.1: Home Base Coordination Protocol

Next the joining node issues `updatetable` requests to those nodes identified that need to be informed of this nodes presence. Afterwards, the joining node tells its predecessor and immediate successor to update their coverage with an `updatecoverage` request. Upon completion of these requests, the joining node is now a part of the system and it begins its stabilization cycle. The stabilization cycle involves a node issuing a `stabilize` request to both its predecessor and successor. The predecessor returns with its predecessor and the successor returns with its successor. So in the event of a successor or predecessor failure, the node will already know of that particular node that will assume the role of the failed successor or predecessor. When the node, home base theater, receives a UANP `put`, `update`, or `delete` naming request, it sends a HBCP `replicate` request to its successor so as to keep the *location maps* and *redundant location maps* consistent.

Universal actors can enter the system at any point after the initial home base theater is up and listening for requests. The universal actors become reachable to other universal actors by issuing a `put`(*UAN*, *UAL*) request to one of the home base theaters.

CHAPTER 4
Performance Analysis

The FHNS prototype has been put through different tests. The intent for these tests is to gather metrics that will support the assertions made in Chapter 2 regarding FHNS being efficient, scalable, and robust. In the first test case, both location-independent and location-dependent names will be used. The results obtained from this first test case will pertain to resilience of FHNS in the event of a home base failure and is used to illustrate robustness. Messaging efficiency and scalability are evaluated in the next test case. The processing time to service requests for location-independent names versus location-dependent names is evaluated. Lastly, a scalability analysis pertaining to the load balancing capabilities is then detailed.

Before going further with the analysis, the reader should consider the specific implementation of the prototype. The tests that have been outlined are intended to measure the performance of the FHNS model; however, the tests predominately relate to the specific implementation of using Chord as the lookup service.

Bearing the impact of Chord upon these results, the analysis of FHNS continues with the first test case regarding robustness[5].

4.1 Robustness

The primary objective with using a lookup service such as Chord is to incorporate the fault tolerance features into FHNS. Since fault tolerance is the primary concern, this capability was tested to establish that the FHNS model is sound. We begin with 8 home bases and fail the home bases one by one, effectively degrading the distributed naming system to a centralized naming system. After each home base failure, `get` requests are made for all the actors in the system to confirm that they are all still reachable. The maximum number of messages it takes to have a request serviced is recorded. For this test, there are an equal number of actor location-dependent names as there are actors with location-independent names.

[5]The tests are conducted using machines with the following specifications:

1. Processor: Pentium (233MHz-1GHz)
2. Network: 100BaseT Ethernet
3. OS / Software: Windows 2000, Java SDK 1.4, SALSA ver 0.5

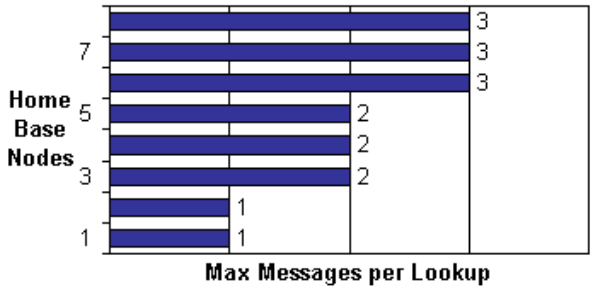

Figure 4.1: Single and Sequential Home Base Failures going from a total of 8 Home Bases to 1.

When the home base of an actor with a location-dependent name fails, FHNS hashes the domain name and port number for that failed home base and determines the immediate successor of the failed home base using a `getnode` request. Once the successor is determined, the request is then forwarded. It is known that the immediate successor of the failed home base has a redundant location map for the failed home base. The immediate successor is thus capable of servicing requests that were intended for its predecessor, which is the failed home base.

Figure 4.1 shows the number of messages it takes to find the back-up home base. The home bases refer to both their immediate location map and their redundant location map when servicing a request and this explains why only one message is needed when two home bases are present. The figure illustrates that FHNS can recover from all home bases failing except for one.

Performance regarding multiple home bases concurrently failing is not evaluated since it is known that this implementation has a single replication depth. Only a home base's immediate successor provides redundancy. If both the home base and its successor failed then there would be actors that would be rendered unreachable. It follows that if the replication depth was doubled, FHNS could recover from two consecutive and concurrent home bases failing. Increasing the replication depth, however, also has the disadvantage of increasing the number of HBCP `replication` requests.

The failed nodes are identified by the home bases periodically performing a stabilization routine. Stabilization involves a home base attempting to communicate with its predecessor and successor. If the home base is not able to communicate it takes the appropriate measures to replace the failed node and transfers and replicates the mapping

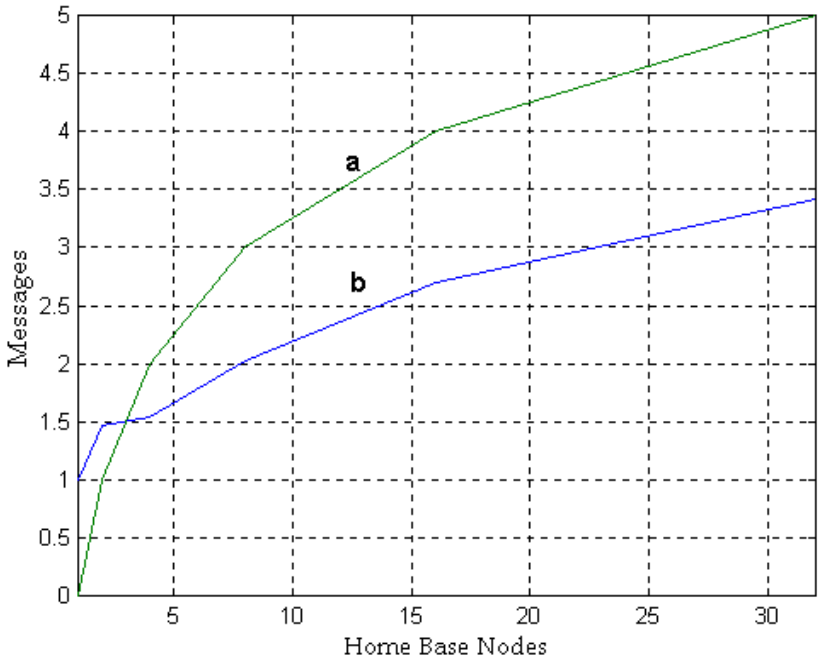

Figure 4.2: Message Count versus Number of Home Bases.

records as needed.

4.2 Messaging Efficiency

Next we turn to the number of messages it takes for FHNS to service a request in a fault-free system. By messaging we count only the requests and forwarded requests and not the responses. When location-dependent names are used only a single message is needed. The messages needed for location-independent names depend on the lookup service employed. Stoica et al [49] state that with high probability, Chord resolves a lookup with $O(log\ n)$ messages. This assertion was confirmed by testing. Figure 4.2 shows the $log\ n$ trend with line a and line b is the average number of messages it took for FHNS, using Chord, to resolve a lookup pertaining to location-independent names. When only two home bases are present, it can take two messages to resolve a lookup; specifically the original request and the forwarded request. In the presence of n home bases, where $(n > 2)$, it takes on average less than $O(log\ n)$ messages.

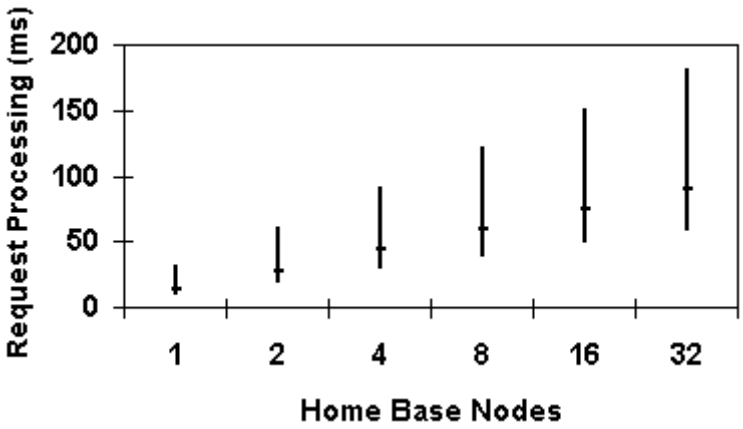

Figure 4.3: Prototype Request Processing Time when Using Location-Independent Names.

4.3 Time Efficiency

FHNS is indeed highly efficient when location-dependent names are used. Its efficiency is put into question with location-independent names. If we do not consider network communication, the *Request Processing Time* (RPT) when using location-independent names is over 3 times that of the processing for location-dependent names. FHNS takes on average 420μs to process a location-independent request and 130μs to process a location-dependent request. This time corresponds to the time elapsed from a home base receiving a request to the generation of a response.

If we consider communication over the network, the total RPT is in the milliseconds. It takes, on average, 14ms for FHNS to service a `put` request that requires no hops be taken. Approximately half this time is spent communicating with the requester and the other half communicating a `replicate` request. When hops are necessitated, the latency is increased by the number of hops. Figure 4.3 shows the minimum and maximum RPTs experienced by FHNS when servicing location-independent `put` requests. Due to these latencies, the location-dependent names are the preferred format for actor names.

4.4 Load Balancing

To fully meet the efficiency and scalability requirements, the home bases can not be too overloaded, else there is the potential for a bottleneck. By using Chord as the lookup service, FHNS is able to attain load balancing. Ideally, load balancing entails that given

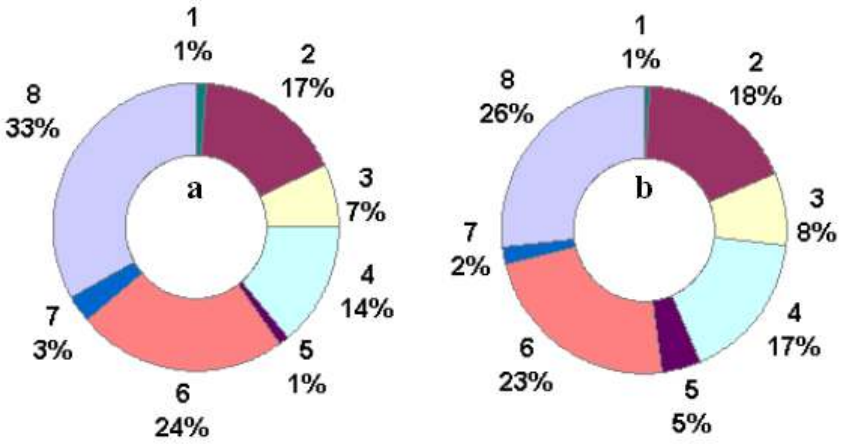

Figure 4.4: (a)The Mapping Records coverage and (b)Namespace coverage divided among 8 Home bases.

K identifiers and H home bases, each home base is responsible for K/H identifiers[6]. In practice, however, the number of identifiers and mapping records per home base exhibits large variations. For instance, when the FHNS prototype consists of 2 home bases, one of the home bases covers 53% of the namespace identifiers and the other home base covers the remaining 47%. In contrast when two additional home bases are added giving a total of 4, 2 of these home bases each only cover 1% of the namespace identifiers, another covers 38%, and the last home base's coverage increases from 53% to 60%.

Figure 4.4 illustrates the namespace identifier coverage and the mapping record coverage. It can be seen from this figure that load balancing is not sufficiently achieved. It could be argued that load balancing is achieved by half the nodes but it certainly cannot be said for them all. Also depicted in the figure is the obvious correlation between namespace identifier coverage and mapping record coverage. The home bases with larger namespace coverage also store more of the mapping records.

The region for a node begins at the node's predecessor's identifier and spans all identifiers up to the node's identifier. Stocia et al [49] propose that each node (home base) divide the namespace into $(O(log\ n)\ +\ 1)$ regions of coverage. In order to do this, the nodes would each have $O(log\ n)$ virtual nodes. Applying this proposal has the effect of "balancing" the coverage amongst the nodes. Figure 4.5 depicts how using virtual nodes equalizes the coverage among the home bases. Although not necessarily ideal, a level of

[6]Assumes all keys are equally "popular" and the nodes are homogeneous.

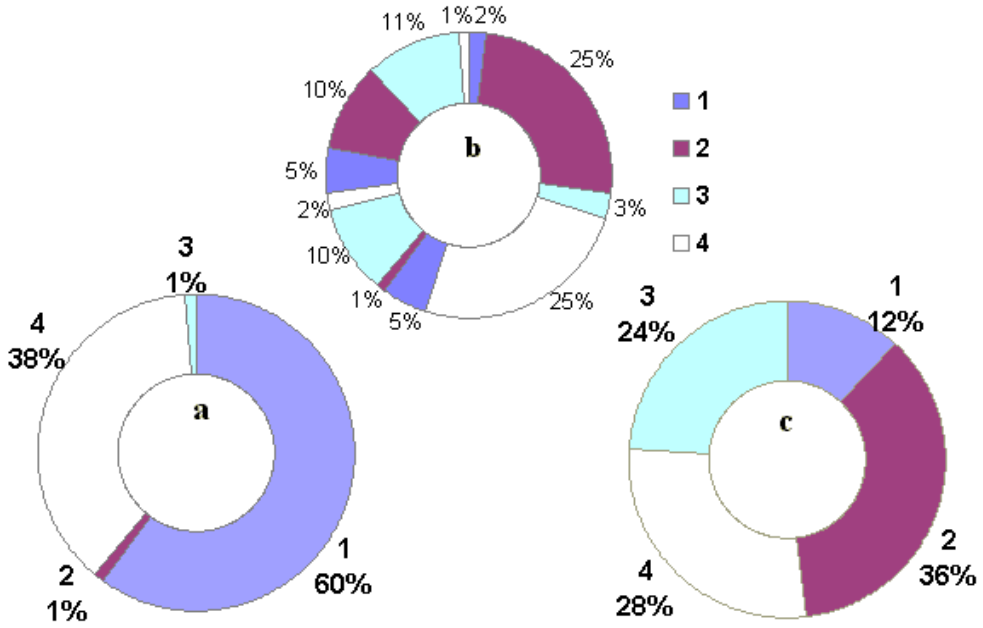

Figure 4.5: (a)Namespace coverage divided among 4 Home Bases, (b)Namespace Coverage divided among 4 Home Bases with 8 Virtual Nodes (c) Total Namespace Coverage with Virtual Nodes.

load balancing is thus achieved when using Chord as a lookup service and having each home base cover $(O(log\ n)\ +\ 1)$ regions. A node's total namespace coverage is the summation of the node's "non-virtual" coverage and its virtual coverage.

Conclusion Drawn from Analysis

The analysis shows that there is a great amount of delay and communication associated with servicing the location-independent names in comparison to the delays and communication associated with location-dependent names. Consequently, it is recommended that location-dependent names be used as a first choice. Furthermore, it is recommended that the fault-tolerant infrastructure that has been established for location-independent names be used for its failover capabilities.

CHAPTER 5
Related Work

The related fields of work discussed in this chapter are categorized into four sections. The first section presents related work on *intentional naming*. Conventional naming services are discussed in section 5.2. Section 5.3 presents different types of naming services used in common mobile agent architectures. Section 5.4 presents other types of peer-to-peer lookup services that could be used by FHNS in lieu of Chord.

5.1 Intentional Naming

Active Names [53], used in WebOS [2], map a name to a chain of mobile programs that can customize how a service is located and also how its results are transformed and transported back to the client. Active Names are application oriented. For example, when an individual accesses a web page from their Portable Digital Assistant (PDA) by using Active Names, the web page being visited can be transformed and resized to fit better on the PDA screen. Active Names are motivated by the observation that end users are no longer interested in the address to contact for a resource, but the resource itself. A similar characteristic shared by FHNS and Active Names is that the names in FHNS determine how an agent is located, which is comparable to an Active Name mobile program customizing how a service is located. Furthermore, these customizations are accomplished transparently to the user.

An intentional name-specifier [22] describes the intent of the application for which the name is being used. The intentional naming architecture uses this name-specifier as part of the naming service it provides. A name-specifier could be `[service=camera] [function=snapshot]`, where contained in the name is the particular service and the desired intent. The name would be resolved by traversing a name tree and reaching the snapshot function. FHNS takes the traditional approach to naming where names are used by applications as a means to reach a location in the network topology. An agent would use FHNS to get the location of an agent that provides a camera service, and then the agent would be able to fulfill its intent by messaging the photographing agent to take a picture.

While active names and name-specifiers can resolve to different resources, as long as they satisfy the user's intent, such an intentional naming scheme would have to be

simulated by FHNS with service broker agents.

5.2 Conventional Naming and Directory Services

The Domain Name System (DNS) [37] translates Internet domain names to Internet Protocol(IP) addresses. This system has already received a lot of discussion throughout this thesis. It is mentioned here again since it is arguably the most conventional naming service presently in use. DNS achieves fault tolerance with the concept of a primary and secondary name server. Clients redirect requests to the secondary server when the primary server has failed.

Windows Internet Naming Service (WINS) [25] is a specific naming service used by Microsoft Windows NT and 2000 Servers. The naming service maps workstation names and locations with IP addresses and it does so without the user or an administrator having to be involved in each configuration change. WINS automatically creates a computer name-IP address mapping entry in a table, ensuring that the name is unique and not a duplicate of someone else's computer name. If the workstation is later moved to another geographic location, the subnet part of the IP address is likely to change. WINS will automatically update the new subnet information in the WINS table. WINS is able to support this "mobility" feature because the name of the workstations are unique not only within the subnet but also within the whole network. Windows 2000 and Windows 98 clients specify more than a single WINS server (up to a maximum of 12 addresses) per network interface. The extra WINS server addresses are used to resolve names only if the primary WIN server fails to respond.

The Network Information System (NIS) [3] is a network naming and administration system for smaller networks developed by Sun Microsystems. An enhancement of NIS, called NIS+, provides additional security and other facilities. Using NIS, each host client or server computer in the system has knowledge about the entire system, which is a permissible characteristic for smaller networks. A user at any host can get access to files or applications on any host in the network with a single user identification and password. NIS is similar to DNS but somewhat simpler and specifically designed for a smaller network. Also similar to DNS, NIS achieves fault tolerance with the use of redundant servers. The use of NIS is expected to wane in the coming years and will eventually be superseded by other naming and directory services such as LDAP [52].

Directory services are comparable to naming services. But whereas a naming service

maps names to locations, the directory service maps names to locations as well as to files, people, services, or other entities.

The X.500 Directory Service [11] is a protocol for managing online directories of users and resources. X.500 is described as a global *White Pages* directory. A user of this service is able to look up people in a user-friendly way by name, department, or organization. Many organizations have created an X.500 directory. Because these directories are organized as part of a single global directory, you can search for hundreds of thousands of people from a single place on the World-Wide Web. Analogous to the tree structure of DNS, the X.500 directory is organized under a common "root" directory in a "tree" hierarchy of: country, organization, organizational unit, and person. An entry at each of these levels must have certain attributes; some can have optional ones established locally. Like DNS, X.500 also provides autonomy. Each organization can implement a directory in its own way as long as it adheres to the basic schema or plan. The distributed global directory works through a registration process and one or more central places that manage many directories. The Lightweight Directory Access Protocol (LDAP) [4] is a "lightweight" (smaller amount of code) version of the Directory Access Protocol (DAP), which is part of X.500. Fault tolerance is achieved by replicating the directory data to multiple locations, and then having a designated primary directory server and associated secondaries.

The Java Naming and Directory Interface (JNDI) [44] is an API specified in Java that provides both naming and directory functionality. Using JNDI, Java applications can store and retrieve named Java objects belonging to any naming service provided that the naming service has a plugged in service provider. The service provider is an implementation of the JNDI Service Provide Interface (SPI). So JNDI is not a defined naming service on its own but rather a tool which makes various naming services available to Java objects. The SPI provides the critical mechanisms for implementing the FHNS API, but it was not used in the prototype. The SPI also supports federation, which is necessary for the home bases to forward requests to one another and distribute responsibility. Presently, LDAP and DNS implementations are available from Sun Microsystems.

5.3 Naming Services in Distributed Systems

Common Object Request Broker Architecture (CORBA) [5] [63] is an architecture and specification for creating, distributing, and managing distributed program *objects*

versus mobile software agents. IIOP (Internet Inter-ORB Protocol) [26] is a protocol that makes it possible for distributed programs written in different programming languages to communicate over the Internet. The CORBA COS (Common Object Services) naming service [51] provides a tree-like directory for object references similar to the way a file system provides a directory structure for files. The naming service enables an object to invoke the services of another object via Remote Procedure Calls. A name is bound to an object through a *bind* or *rebind* operation and unbound via an *unbind* operation. The object is then later found through a *resolve* operation. CORBA is widely used for distributed computing between heterogeneous platforms and its services are used by mobile agent platforms. It is worth pointing out that the operations used for naming are analogous to the one presented for FHNS. But unlike the home bases of FHNS, CORBA name servers exist as single points of failure. The name servers in CORBA maintain name-to-object mappings. The objects are rendered unreachable when the respective name server is unavailable. Realizing that this is a problem that needs addressing, efforts have been made to make the CORBA name service fault tolerant. Electra [32] provides an infrastructure for users to develop reliable distributed computing applications, including an adaptive component for dynamic updates. Electra is built using CORBA, and addresses a number of issues resulting from CORBA's inherent inability to provide reliable distributed communication.

The 2K global naming service [27] is a naming service for use in the 2K Operating System [23]. The 2K OS is based on the CORBA middleware and is described as being a distributed, object-oriented, network-centric and adaptable operating system. The 2K naming service builds upon and enhances the COS naming service. The 2K naming service provides a global namespace with a global root and it also provides a model for administering the namespace. The 2K naming service, however, does not address the single points of failure inherent with the COS naming service.

Java Remote Method Invocation (RMI) [48] enables an object (client) to invoke the methods belonging to another object (server) running on a different JVM. The client object obtains a reference to the server object via the RMI *registry*. The registry functions as a name server by keeping a name to remote object mapping, where the remote object mapping includes the network location of the object. A server object registers a name with the registry using a `bind` or a `rebind` method call. The client obtains a reference to the remote object by using the name of the object in a `lookup` method

call. For instance, assume a registry is running on host.net and listening for requests to come in over port 1099. A server object, called *cameraObject*, registers itself by issuing a `rebind`(*//host.net:1099/camera*, *cameraObject*) request. The full name, or absolute name, of the server object is *//host.net:1099/camera*. The names are virtually identical to the location-dependent names of FHNS, where the registry servers can be likened to home bases. The client object issues a lookup(*//host.net:1099/camera*) and receives a reference to the *cameraObject* that can be used to invoke methods belonging to this object. If host.net experienced a power failure and was shut off, the *cameraObject* would no longer be accessible. This is another case where the name servers exist as a single point of failure. Redundancy can be achieved by a server object registering with multiple registry servers. This approach entails that the clients have a priori knowledge of all these registry servers. Furthermore, the client has to identify when a registry server has failed and be able to recover accordingly. Compare this approach to FHNS, where failover happens transparently to the clients. Filterfresh [6] uses another approach in addressing these single points of failure and builds a fault tolerant infrastructure with the use of FT registries. FT registries are a group of replicated RMI registry servers. Filterfresh extends the RMI interface by introducing a `multibind` method. The mapping from server name to remote object is stored at numerous registry servers. Filterfresh allows for new FT registry servers to join a group to provide for more redundancy. With FHNS, new name servers do not need to be added to provide for more redundancy. FHNS uses existing name servers in the name identifier space (i.e. identifier circle with Chord), where every node has a corresponding backup. If more redundancy is desired, the number of backups per node can be increased to span across more nodes within the name identifier space and so the addition of new servers is not needed. Taking an approach similar to Filterfresh, the RMI registry servers could be extended to serve as FHNS home bases. Doing so will have less of an impact on those mobile agent systems presently using RMI that want to incorporate FHNS.

The Globe Location Service (i.e. naming service for mobile agents) [54, 55] maps object identifiers to locations to support mobile objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative regions, effectively constructing a static world-wide search tree analogous to DNS. In its most basic organization, the Globe Location Service does not scale since the higher level directory nodes in the search tree have to handle a large number of requests and they also have high storage demands.

The authors propose that the directory nodes be partitioned such that each subnode is responsible for a subset of the records originally stored at the directory node. However, even with this solution, it has yet to be shown that the service is scalable in terms of servicing requests. Fault tolerance is supported by the directory nodes in the same way that DNS name servers are replicated.

There are many mobile agent platforms that are in development to support distributed computing. Mentioned here are a few representative ones. Chapter 1 identified coordination mechanisms that could be used by interacting mobile agents. The blackboard-based coordination model, for example, is used in the Ambit [13] formal model for mobile computation. And the meeting-oriented model is used by the MOLE [7] mobile agent system. This thesis was centrally focused on the direct coordination model and the mobile agent platforms that use this type of coordination.

Java is characterized as being an object-oriented, distributed, interpreted, robust, secure, platform-independent, portable, multithreaded, and dynamic language. Due to these characteristics there are many mobile agent platforms that are Java-based. These Java-based agent systems can exploit the RMI mechanism to transparently invoke a method of a remote object and as such a direct coordination mechanism is already provided. What follows are descriptions of Java-based mobile agent systems.

The World-Wide Computer [56] was described in detail in Chapter 3. An FHNS implementation has been developed for use by the WWC to address the single points of failure with the dedicated name servers.

Concordia [28], Odyssey [39], and JAFMAS [15] are all agent systems that use RMI and serialization for agent mobility and messaging. The registries of RMI, exist in these systems as single points of failure. Agents become unreachable when their corresponding registry server fails.

The Agent Naming System (ANS) [38] is a specific naming system intended for use by the Aglet system of mobile agents [64, 33]. ANS associates names with places as opposed to having names associated with aglets. An aglet then uses ANS to move from one place to another by referring to the names of those places. ANS facilitates mobility but doesnt explicitly address how the aglets locate each other for the purposes of communication. Presumably this service is meant to be used in conjunction with an existing naming service for aglets. The Aglet system is Java-based and it uses standard communication and naming mechanisms in place, specifically RMI and CORBA.

The Web Agent-based Service Providing (WASP) [67] project consists of mobile agents that move across web servers. The agents have URL-style names that are used to locate the agents. An agent begins its tasks at a particular web server and when it moves to its next location, it leaves behind a pointer that it continuously notifies when it relocates. The web server storing this pointer can be called a "home base". Other agents use these pointers to obtain references to those agents with whom they wish to interact. Once again, a single point of failure exists. When the "home base" web server fails, the pointers are gone and the respective agents are not accessible.

The Cognitive Agent Architecture (Cougaar) [65] is another Java-based architecture for the construction of large-scale distributed agent based applications. Cougaar includes a distributed naming service for finding other Cougaar agents. The current implementation uses JNDI over RMI to a single naming server.

Single points of failure exist in all the direct coordination mechanism employed by these agent systems. The Fault-Tolerant Home-Based Naming Service addresses this very problem, without compromising lookup efficiency in the general case.

5.4 Decentralized Lookup Services

5.4.1 Peer-to-Peer Lookup Services

Gnutella [66] and Freenet [17] are peer-to-peer file sharing programs that provide a lookup service for locating files. In Gnutella, requests for a file are flooded with a certain scope, which effectively means there could be files available that are not found because they are outside the scope. Flooding on every request is not scalable and the service is not capable of giving a definitive success or failure for a given lookup. A naming service must be capable of giving a definitive success or failure and so the flooding approach was never considered an option in locating mobile agents. Freenet also does not give definitive successes or failures.

JXTA [20] is a Java-based technology that provides a set of open protocols that allow any connected device on the network to communicate and collaborate in a P2P manner. To achieve this goal, the JXTA platform is composed of classes and methods for managing and transmitting application and control data between JXTA compatible peer platforms. One protocol that relates to decentralized lookup services is the Peer Resolver protocol. This protocol allows a peer to send and receive generic queries to find or search for peers, peer groups, and other information. A *resolver*, from a JXTA perspective,

resolves a question to an answer. The resolver should be thought of as a network-wide query. Instead of specifying a single peer, a group of peers is queried. The messages of the resolver are quite simple. The messages, however, are not guaranteed to reach their destinations nor are the results, if they exist, guaranteed to arrive back at the source of the query. So, given this acknowledgement, the lookup service presently provided by JXTA is like the lookup services provided by Freenet and Gnutella in that definitive successes or failures are not given.

5.4.2 Distributed Hash-table Lookup Services

Chord [49] has received a lot of attention in this thesis already, it is presented here again to discuss other Chord-based naming services that have been developed and evaluated. One such implementation is called DDNS [18]. DDNS stores and retrieves DNS records using a Chord-based distributed hash table. The system could accept conventional DNS queries, perform the appropriate Chord lookup, and then send a conventional response. An evaluation of the system showed that that the system adequately achieved load balancing. The evaluation also showed that there is a large latency introduced with using DDNS over DNS, it takes nearly 8 times as long to service a request. Another peer-to-peer DNS implementation was developed to address Denial of Service (DoS) attacks [58]. This later implementation demonstrated that Chord can provide the fault tolerance needed to prevent a Denial of Service.

The following lookup services are similar to Chord in many respects. In the later two cases, the lookup services yield better performance than Chord.

CAN [42] is described as being a distributed infrastructure that provides hash table like functionality on an Internet scale. This infrastructure is specifically used to map keys to values as is done with consistent hashing and Chord. Unlike Chord, CAN uses a d-dimensional identifier space. Strings are hashed for each dimension to give a corresponding coordinate in all such dimensions. CAN routes in $O(d*n^{1/d})$ messages. In order to achieve logarithmic scalability, CAN needs to set $d = log\ n$, which requires that the number of nodes n be known in advance. CAN does not appear to offer any advantages over Chord.

Tapestry [60], as used in the OceanStore [31] wide-area storage system and also used in the Bayeux [61] wide-area data dissemination architecture, is a location routing infrastructure that provides location-independent routing of messages. Tapestry maps nodes and names to identifiers and uses suffix-routing when performing a lookup. For

instance to route from the node with an identifier 5324 to a node with identifier 0629, the path traversed would look something like 5324 $\rightarrow$ 234**9** $\rightarrow$ 14**29** $\rightarrow$ 7**629** $\rightarrow$ 0629. Each node maintains a routing table with $b * log_b\ n$ entries at each level, where b is the base of the identifiers. Redundancy is used between neighboring nodes for fault tolerance. The routing mechanism is also fault tolerant since nodes can route around a failed node. Compared to Chord, this *hypercube* type of routing is arguably more complex but still effective in resolving a lookup in $O(log\ n)$. The performance achieved with a lookup in Tapestry match those available with Chord.

Pastry [46], used in the SCRIBE [47] event notification infrastructure and also used in the PAST [21] peer-to-peer storage utility, is similar to the hypercube routing used by Tapestry. Pastry, however, is able to resolve a location in $log_{2^b}\ n$ messages, where b is a configuration parameter with a typical value of 4. The routing table for nodes using Pastry contain up to $log_{2^b}\ n$ rows with $2^b - 1$ entries each. The nodes also maintain additional routing information: a neighborhood set and a leaf set. The neighborhood set contains $|M|$ entries and the leaf set contains $|L|$ entries, where typical values for $|M|$ and $|L|$ are 2^b and $2 * 2^b$. The routing tables are more complex than the routing tables for the other lookup services. This complexity would be most reflected in maintaining accurate routing information and in the initialization of the table when a node joins the system. However, with a more complex routing table comes better performance in terms of efficient lookups. An FHNS implementation using Pastry will yield better performance with respect to lookups.

SPRR, Simple Plaxton, Rajaraman, and Richa [41, 34], is another distributed data lookup scheme. Of all these service, SPPR is most similar to Chord. One notable difference is that an SPRR node has entries in its routing table for both its forward neighbors in the logical ring as well as its backward neighbors. Another difference is that when a node leaves or joins in Chord, $O(log^2\ n)$ messages could result. SPRR performs all operations for node-joins, node-leaves, and lookups in $O(log\ n)$ messages. An FHNS implementation using SPRR would give better performance in a highly dynamic system with nodes joining and leaving frequently.

CHAPTER 6
Future Work

6.1 Optimizations

There are disadvantages to FHNS in regards to location-independent naming. As already identified, caching could be employed to better the performance. Another drawback with using a location-independent mechanism is the upkeep of the routing table information. Chord specifies that a stabilization algorithm be performed periodically, which introduces overhead. The prototype performs stabilization every couple of seconds, which could be too much in some cases or not enough in other cases. More research and testing is needed in coming up with an adequate and efficient stabilization algorithm.

6.2 Security

From researching other name services it becomes apparent that security and naming are two problems that are addressed separately. This most likely is due to the fact that security is a very broad subject that has far-reaching ramifications. There are numerous issues pertaining to security in a distributed computing framework. The mobile agents themselves have to be kept secure from malicious hosts and malicious agents. The hosts have to be kept secure from malicious agents. Naming service security pertains to data integrity and there are solutions in place to maintain data integrity such as public-key encryption schemes as was discussed in Chapter 3.

Another problem pertaining to security is too much restriction as is experienced with firewalls. This too is to be addressed in the security research efforts.

6.3 Publish and Subscribe Model

The publish-subscribe paradigm is not presently employed by FHNS. The publish-subscribe paradigm would entail that a home base subscribe to another home base that informs the subscribing home base when an agent they are tracking on behalf of another agent relocates. The cached references would be kept current by a *pushing* mechanism. So say GYPSY queried HomeBase1 for NOMADS location. HomeBase1 determines that HomeBase2 is the home base for Nomad. HomeBase1 subscribes to HomeBase2 with respect to NOMAD. Each time NOMAD relocated, HomeBase2 would notify HomeBase1.

If NOMAD moved on average 40 times in a certain time frame and GYPSY communicated with NOMAD on average 2 times in the same time frame, the publish-subscribe model would yield 10 times the amount of messages than the present version of FHNS using location-dependent names. Conversely, however, the publish-subscribe model would yield less messaging than the present version of FHNS if GYPSY communicated with NOMAD more frequently than NOMAD relocated. But there is no way to determine at a comprehensive level what the participating agents will do more frequently: communicate or relocate. Currently, FHNS is driven by need: home bases *pull* the information they need and only when the need arises. Future research will determine if a publish-subscribe service should be incorporated into FHNS and to what degree.

6.4 Multiple Namespaces

One of the main features of DNS is autonomy. The system is designed to allow each organization to assign names to computers or to change the names within their domain without having to inform or seek the approval from a central authority. FHNS should likewise permit autonomy in terms of namespace identifiers. Organizations could then construct their own namespaces much in the same way that they extend their respective domains. The namespace identifier could be a key belonging to the global namespace, effectively giving a ring of rings. The key would map to a node or nodes belonging to the organizational namespace so that agents could communicate between namespaces. location-independent names facilitate multiple namespaces and could be extended to look something like

$$urn :< Global\ Namespace > \_ < Organizational\ Namespace >:< NSS > .$$

Location-dependent names pose a problem when home bases fail since the namespace that the failed home base belonged to is not known, and consequently the successor for the failed home base can not be determined. Multiple rings reintroduce the problem of agents becoming unreachable. Solutions to this specific problem and other potential problems are a topic of ongoing research [60, 46, 42].

6.5 Garbage Collection

Garbage collection is a seemingly divergent topic to naming services. However, distributed garbage collection (DGC) is an interesting problem. The question to be answered

is how does a system know when an agent is garbage? In the WWC, for example, universal actors are not garbage collected and resources are never recovered because other actors can create references to the universal actors from their UANs. Some of the DGC solutions that have been presented involve a "stop the world" approach where the system is analyzed and all the agents that do not have a reference being made to it from some established root are said to be garbage and are later collected. FHNS provides a mechanism that could be used instead of a "stop the world" approach. Home Bases could also serve as distributed garbage collectors. The Home Bases can know when an agent has references made to it by tracking the specific `get` requests for a given agent. Agents that have not had a reference made to it within a given number of time frames could be a candidate for collection [43].

CHAPTER 7
Conclusions and Contributions

The Fault-Tolerant Home-Based Naming Service is a novel approach to locating mobile agents in a distributed computing framework. FHNS builds upon DNS and a peer-to-peer lookup service. And as a result of this meshing, this naming service is fault-tolerant as was shown by using a lookup service such as Chord. This naming service is also very efficient in resolving the location of a mobile agent.

Contributions of the Thesis

The specific contributions made in this thesis are as follows:

1. **A naming service specifically designed for use by mobile agents**. From the research that has been done, it was observed that efforts on the specific topic of naming services for mobile agents has not received much attention, at least when compared to mobile agent security, mobility, and messaging. Also from researching the naming services in current mobile agent platforms, it was observed that efforts do need to be focused on this very topic. Most Java-based agent systems use RMI or Corba which introduce single points of failure.

2. **A naming service that is highly efficient, fault tolerant, and scalable**. Location-dependent naming services, such as the one found in RMI for instance, are efficient in resolving a name to a location but introduce a single point of failure. FHNS is as efficient as these location-dependent naming services and it is also fault tolerant. Furthermore, the fault tolerance extension does not degrade the efficiency in resolving location-dependent names until a failure occurs.

3. **A naming service with more extensive fault tolerance than the conventional redundancy model**. Fault tolerance is a critical requirement with a naming service. Traditional redundancy schemes involve a designated primary server and an associated secondary server. The secondary server, however, does not necessarily have a back up. So when the primary fails, the system can no longer be described as being fault tolerant because the secondary, or the new primary, server can not fail. With FHNS, each server has a backup and each server is a backup for another

server. The servers can fail down to a single remaining server and FHNS will still work as needed.

There are numerous mobile agent systems currently in development. Many of the systems are Java-based and have agents that communicate directly with other agents. Direct communication requires direct coordination mechanisms. From researching numerous mobile agent systems, it was observed that the naming mechanisms currently employed by these same systems are not as efficient as they could be and/or they are not robust. FHNS could be incorporated by these mobile agent systems to realize an efficient and robust naming mechanism.

Literature Cited

[1] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] Mike Dahlin Eshwar Belani David Culler Paul Eastham Amin Vahdat, Tom Anderson and Chad Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, 1998.

[3] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *International Workshop on Services in Distributed and Networked Environment*, pages 84–91, 1996.

[4] Brian Arkills. *LDAP Directories explained: an introduction and analysis.* Addison Wesley Longman INC, 2003.

[5] Henry Balen. *Distributed Object Architectures with CORBA.* Cambridge University Press, 2000.

[6] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik. Filterfresh: Hot replication of Java RMI server objects. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 65–78, Santa Fe, New Mexico, 1998.

[7] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole - concepts of a mobile agent system. Technical Report TR-1997-15, 1997.

[8] Tim Berners-Lee. Uniform Resource Locators (URL). Informational RFC 1738, 1994.

[9] Tim Berners-Lee. Uniform Resource Identifiers (URI): Generic Syntax. Informational RFC 2396, 1998.

[10] Marc Branchaud. A Survey Of Public-Key Infrastructures. Master's thesis, McGill University, Montreal, 1997.

[11] S. Heker C. Weider, J. Reynolds. Technical Overview of Directory Services Using the X.500 Protocol. RFC 1309, 1992.

[12] G. Cabri, L. Leonardi, and F. Zambonelli. How to Coordinate Internet Applications based on Mobile Agents. In *Proc. 7th IEEE Workshops on Enablings Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 104–109, Stanford, CA, 1998. IEEE Computer Society Press.

[13] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[14] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[15] D. Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, University of Cincinnati, 1997.

[16] S. Cheung. *An Intrusion Tolerance Approach for Protecting Network Infrastructures*. PhD thesis, University of California, Davis, 1999.

[17] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[18] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using a peer-to-peer lookup service. In *IPTPS*, 2002.

[19] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service, 2001.

[20] Darren Govoni Juan Carlos Soto Daniel Brookshier, Navaneeth Krishnan. *JXTA: Java P2P Programming*. Prentice Hall PTR, 2002.

[21] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[22] Hari Balakrishnan Elliot Schwartz, William Adjie-Winoto. An Architecture for Intentional Name Resolution and Application-level Routing. Technical Report MIT/LCS/TR-775, 1999.

[23] M. D. Mickunas K. Nahrstedt F. J. Ballesteros. F. Kon, R. Campbell. 2K: A Distributed Operating System for Heterogeneous Environments. Technical Report UIUCDCS-R-99-2132, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.

[24] G. H. Forman and J. Zahorjan. The challenges of mobile computing. Technical Report TR-93-11-03, IEEE Computer, 1993.

[25] Desmond Fuller. *Windows NT Browsing WINS and Directory Service*. Macmillan Computer Publishing, 1998.

[26] Michael Hittesdorf. *CORBA IIOP Clearly Explained*. Morgan Kaufmann Publishers INC, 2000.

[27] Muhammad Ziauddin Hydari. Design of the 2K Naming Service.

[28] M. ITA. Concordia: An infrastructure for collaborating mobile agents.

[29] B. Joy J. Gosling and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[30] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[31] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[32] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[33] D.B. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Java Aglets*. Addison Wesley Longman INC, 1998.

[34] Xiaozhou Li and C. Greg Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 82–89. ACM Press, 2002.

[35] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[36] R. Moats. URN Syntax. Informational RFC 2141, 1997.

[37] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, 1987.

[38] Tomasz Muldner and Thian Tin Ter. Building Infrastructure for Mobile Software Agents. *World Conference on the WWW and Internet WEBNETC*, 2000:419–424, 2000.

[39] Brian D. Noble, Morgan Price, and Mahadev Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. Technical Report CS-95-119, Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing, 1995.

[40] Ian Piumarta, Marc Shapiro, and Paulo Ferreira. Garbage collection in distributed object systems. In *Workshop on Reliability and Scalability in Distributed Object Systems, OOPSLA'95*, Austin, TX, 1995.

[41] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[42] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[43] Helena C. C. D. Rodrigues and Richard E. Jones. A cyclic distributed garbage collector for Network Objects. In Ozalp Babaoglu and Keith Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, pages 123–140, Bologna, Italy, October 1996. Springer.

[44] Scott Seligman Rosanna Lee. *JNDI API Tutorial and Reference: Building Directory-Enabled Java Applicaitons*. Addison Wesley Longman INC, 2000.

[45] V. Roth. Scalable and secure global name services for mobile agents. In *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, 2000.

[46] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[47] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[48] Prashant Sridharan. *Advanced Java Networking*. Prentice Hall PTR, 2002.

[49] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[50] Karim Taha and Thomi Pilioura. Agent Naming and Locating: Impact On Agent Design. Technical report, 1999.

[51] William A. Ruh Thomas J. Mowbray. *Inside CORBA*. Addison Wesley Longman INC, 1997.

[52] Michael Haines Tom Bialaski. *Solaris and LDAP Naming Service: Deploying LDAP in the Enterprise*. Prentice Hall, 2001.

[53] Amin Vahdat, Michael Dahlin, Thomas E. Anderson, and Amit Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[54] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1 1998.

[55] Maarten van Steen, Franz J. Hauck, and Andrew S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of the TINA'96 Conference*, 1996.

[56] Carlos Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[57] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[58] Min Wu and Omar Bakr. Peer-to-Peer DNS infrastructure against DoS attacks. www.mit.edu/ minwu/download/6829.pdf.

[59] Fips 180-1. secure hashing standard. U.S. Department of Commerce/NTIS, National Technical Information Services, April 1995.

[60] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[61] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.

[62] Estimates on the Size of the Internet. http://www.netbook.cs.purdue.edu/othrpags/qanda262.htm.

[63] Common Object Request Broker Architecture. http://www.omg.org.

[64] Aglet Workbench from IBM. http://www.trl.ibm.co.jp/aglets/index.html.

[65] Cognitive Agent Architecture (Cougaar). http://www.cougaar.org/.

[66] Gnutella. http://gnutella.wego.com/.

[67] The WASP Project Approach. http://wasp.cs.vu.nl/wasp/.