Agent Usage Patterns: Bridging the Gap Between Agent-Based Applications and Middleware

Eugene Hung*
Joseph Pasquale

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092
{eyhung,pasquale}@cs.ucsd.edu

Technical Report

November 17, 1999

Abstract

The concept of agents—programs that are capable of transporting themselves across a [heterogeneous] network to execute and return results—is a fascinating if troubled area of research. While theoretical advantages of agents have been well-established, few agent-based applications have been commercially successful. We argue that the lack of applications stems from a lack of understanding essential agent usage patterns. In this paper, we identify a set of fundamental patterns that support the design of agent-based applications that scale performance, reliability, and security. To evaluate their performance, some of these patterns were implemented in Java to demonstrate customizable and scalable performance.

*Supported by a fellowship from National Semiconductor

List of Figures

1 Traditional (RPC) implementation vs. Agent implementation 8
2 Agent monitoring a space probe far away from the client . . . 11
3 Decoupling computation to bypass an unreliable link 13
4 Customizable search controlled by host to ensure privacy . . . 14
5 Agent monitoring stock quotes 17
6 Structure graph key 33
7 Structure of the Bypasser pattern 34
8 Structure of the Commuter pattern 35
9 Structure of the Isolator pattern 36
10 Structure of the Monitor pattern 37
11 Structure of the Interface pattern 38
12 Structure of the Rover pattern 39
13 Structure of the Shepherd application 41
14 Structure of the Trader application 42
15 Structure of the RoboTrader application 43
16 Three Network Paradigms 44
17 Experimental Setup 47

List of Tables

1 Comparison of Response Time for Various Paradigms 45
2 Success Rates for the RoboTrader vs. the Trader 48

1 Introduction

The idea of mobile code or *agents* (programs that can move to different locations and execute) is promising : agents offer a faster, more flexible, more reliable, and more secure alternative for network communications. The idea of agents is not new [18], and yet, agents have failed to make a measurable impact in today's programming environment. Most network applications, including the ones that would significantly benefit from agents, still communicate using traditional client/server methods. In this paper, we introduce tools called *agent usage patterns* that will aid in the development and use of agent applications, ultimately promoting the development of network applications that benefit from agents.

1.1 Next-Generation Network Applications

Troutman's Third Law states that "Any useful program will have to be changed.", and network applications are no exception. This constant evolution results from the demand for more features or solutions. The next generation of network applications (NGNAs) are those that can be visualized using today's technology but not yet implemented in a widespread fashion. Examples of NGNAs include remote multimedia filters, stock quote monitors, and virtual worlds. A goal of this work is to stimulate the development of NGNAs, so this section will survey several NGNAs and identify the obstacles facing their implementation.

1.1.1 The Applications

The Electronic Marketplace Sparked by the success of the World Wide Web (WWW), the Internet is developing into a global marketplace. The size of this marketplace is unprecedented in history, as is the corresponding variation in demand: different people desire different treatment. However, the primary reason behind these large-scale problems — the ability to transcend the limitations of physical location via computers — also provides the theoretical means to address them, through clever programming.

Given this, one might wonder why the electronic marketplace, as it currently stands, depends so much on traditional mail-order methods of service listing and delivery; it would seem as if the flexibility inherent in computing would allow a flexibility in marketplace transactions unknown till now. For

example, when purchasing a service, one is bound to the services that the seller had the foresight to offer; if the buyer wants a special service, he is out of luck. Traditional client/server methods of computer communication do not allow service customization without requiring much effort and cost on both sides to establish a specialized protocol.

Virtual Worlds A common theme in speculative fiction is the concept of the virtual world on top of an information network [13, 30]. The virtual world is a representation of data objects in a 3-D graphical environment, as if the computer were a window on another world. Several advantages can be gained from such an approach: improved user interaction, extra information from using the third dimension, and a more natural method of control [6].

However, current virtual world implementations suffer from many problems, such as clumsy, intractable interfaces; expensive equipment [25]; failure to scale well as bandwidth escalates [7]; and "lag" from latency penalties for updates. In particular, since users experience motion sickness if objects are not updated in a realistic manner [25], the lag resulting in part from the existing network infrastructure forms a major obstacle to creating successful virtual worlds.

Flexible databases Another NGNA revolves around the protection of intellectual property [2]. Many databases ideally want to maintain some amount of control over the information dispensed, but the ease of duplicating digital media renders this impossible once a client gets its hands on the information.

One attempt at maintaining the privacy of intellectual property is to have the server perform client-requested operations upon its private data, and sending out only the results. This process thwarts any attempt to duplicate the original data, as the information never leaves its home site. The drawback of this solution is that, under traditional communications methods, unless a server can anticipate or completely understand a client's request, it cannot support a specialized, complex operation upon the private data. What is needed is a middle ground whereby the server can keep control of the data without restricting the client's ability to process the data.

Mobile applications In the fast-paced environment of today's business world, more and more people are finding it necessary to be able to access

computing services wherever they go. Current networking technology has succeeded in bringing the network wherever radio signals can reach, at the price of less-reliable connections and uncertainty about resources (such as bandwidth).

Common desktop applications translate poorly to a wireless environment. In particular, applications that demand reliable, continuous data communication (e.g., Web browsers, file transfer) work poorly in a mobile environment unless specific, expensive measures — such as redesigning the network to capture state — are taken. Easily porting these applications to a mobile environment requires a general, widespread, and cost-effective mechanism for maintaining reliability, which does not exist.

One can also apply the same argument to applications which depend on a certain level of resource availability and capacity to function correctly. Here, what is needed is a way to efficiently and transparently compensate for unexpected shortages of resources, otherwise known as adaptation. The ability to adapt to the environment is extremely valuable in mobile applications. Some research on adaptation has focused on support from the operating system [24, 27]. However, a general, cost-effective mechanism for supporting adaptation at the application level would significantly reduce porting problems for a generic desktop application, and, like a general mechanism for maintaining network reliability, does not exist.

Resource sharing The explosive growth of microprocessor performance over the last few decades has created an abundance of computing resources. It is common to see workstations with power equivalent to the supercomputers of a few years ago sitting idly on their owners' desks. Despite these advances, there still exist applications that would either benefit from or require more resources than those provided by a single machine, especially in a resource-poor environment (such as a network computer).

Researchers have attempted to harness the idle resources of other computers by implementing global resource managers at the network level: these detect idle machines to help other machines [23, 1]. However, there are situations where it would be better for the application itself to determine and use these resources. This would require a means to access remote environments and monitor them continuously for idle resources, without jeopardizing the security of the remote environment. Unfortunately, most research in this area has ignored this problem by assuming only local network operation.

This severely limits the scope and flexibility required for demanding applications, but is necessary given the constraints of the client/server model: if the operation is extended to any remote network, the latency problems involved in monitoring and security problems involved in resource access from a foreign site are immense.

Flexible distributed control In a distributed system, it is often hard to manage and control resources spread out over the network, yet there is a large demand for applications that can do this efficiently [4]. For example, take the common problem scenario of monitoring and maintaining the systems sharing a network. Current management techniques use a protocol such as SNMP to provide and update network information. The simplicity of SNMP has led to its widespread adoption, yet it has been described as a "band-aid" due to its failure to provide adequate security measures and inability to perform high-level tasks [34]. A government-backed attempt to introduce a more powerful, secure protocol (CMIP) failed due to poor performance, as the existing client/server infrastructure could not support it in an efficient manner [34]. A flexible, powerful, secure, and efficient approach to distributed control is needed.

Multimedia filters Playing back multimedia streams requires synchronization of the different media streams and significant amounts of bandwidth. Certain clients, facing limited or uncertain bandwidth/CPU resources, may desire a lower quality product in order to meet synchronization requirements. For example, a client watching a video on a wireless computer may want to cope with its more limited and unreliable resources by having the video playback program, or "player", sacrifice image fidelity or frame rate. However, most current players are not designed to filter the incoming stream to cater for the special needs of the client and its environment. Even the players that are able to flexibly respond to the client's needs through filtering have performance problems, usually as a result of network delays between the application and the server [8].

The ideal player should be able to adapt quickly to changing conditions. Note that this is similar to mobile communications, but deserves special mention because real-time constraints may result in a low Quality of Service(QoS) even over reliable networks. Furthermore, the ability to vary the resolution of multimedia content lends itself to adaptation.

Real-time monitors Certain applications may require real-time operations to be performed at a very distant site. The archetypical example is that of a space probe which needs instructions on how to adapt to changing conditions [18]. Use of a communication channel between the probe and its source will be ineffective due to the staggering delays from latency. Such long-distance control requires a way to avoid the latency costs in order to facilitate real-time interaction.

A useful everyday application that uses this concept is one which monitors a remote stock quote site. Most current stock monitors are passive, relaying information over the network back to the owner for a decision. However, if the latency and packet processing overhead significantly delays the transaction, the client may find that the option to buy or sell may no longer be available, especially during periods of unusually heavy trading. It would be highly advantageous if the monitor program itself could be empowered to make the decision, thus eliminating delays and correspondingly, the client's frustration with unconsummated transactions.

1.1.2 Analysis of Applications

The NGNAs described above share many of the same problems. These are:

Inability to adapt The fluctuating conditions of a mobile environment, the remote monitoring and acquiring of resources, and the variable user requirements of a multimedia player may all benefit from a more general adaptation mechanism. Such a mechanism should be flexible enough to support user-defined resources for tracking, and also let the user specify when and how to react.

Inflexible user interfaces The electronic marketplace, flexible databases, and multimedia players all rely on the ability for the user to have more control over the services offered. Instead of being a passive receptor of services, the user — or a third-party developing software on behalf of the user — should be able to create customized environments that directly cater to the user's needs without unnecessary complexity. This last point, complexity, is often ignored in theory, but is important in practice. For example, a VCR's advanced features are useless if its user cannot operate them.

Excessive latency The virtual world, the real-time monitor, and the multimedia filter have problems with network latency costs significantly affecting their performance. The escalating use of the Internet will only exacerbate this problem as more and more messages will face potential queuing delays from congestion. Even in an ideal world with no congestion and unrivaled power on every desktop, the latency problem will remain, as the speed of message delivery is bounded by the speed of light.

Susceptibility to failure Many applications perform at an acceptable level in an ideal environment, but deteriorate rapidly in a environment which cannot provide resource or packet delivery guarantees, such as a wireless network. Without the ability to adapt, wireless machines will not be able to perform well compared to their wired counterparts.

Lack of security Some of the above applications are merely flexible extensions to existing applications (resource sharing, network management, flexible databases). These applications are untenable in the current environment due to their inability to cope with violations of system security. For example, it is foolish to give power over system resources to a remote, untrusted system for resource sharing or network management, nor is it wise to expose one's assets (the data set of a database) to a customer who can easily dilute said assets through duplication.

Until these problems are addressed, the full potential of these NGNAs will not be realized.

1.2 Agent-based Computing

The problems raised by the previous section's analysis can be categorized into four areas: flexibility (which includes adaptivity), performance (latency), reliability (failure), and security (privacy and integrity). A communications model based on agents specifically provides these advantages : programmability of agents for flexibility, latency reduction for performance, encapsulation of state for reliability, and built-in support mechanisms for security.

We define agent-based computing as a mechanism for communication that relies on programs which, in the course of operation, can move to another machine and execute. These programs are called *agents*, because they carry

out orders on their owner's behalf. While the term *agent* is rapidly becoming overloaded in computer science, particularly in the subfield of artificial intelligence [11], any future reference to agents will be under this definition.

1.2.1 The Electronic Marketplace Revisited

To clarify how agents operate, the implementation of the electronic marketplace scenario is compared between traditional and agent-based techniques. A more detailed description and analysis of agents is presented in the following section.

Traditional Implementation Under the traditional client/server model, an electronic transaction is carried out by having the buyer (the client) and seller (the server) communicate using remote procedure calls (RPC) [5] However, the flexibility of client applications is usually restricted to choice of server and to the options pre-defined by each specific server. To take a concrete example, a user who wishes to find and purchase a book can only choose the remote bookserver (e.g., Amazon vs. Barnes & Noble). Once there, the user is forced to use whatever search mechanism the server provides. Customized operations for each client, such as a specialized search (e.g., by publishing house) are unavailable unless the server has anticipated the client's wishes—a condition that is neither practical nor scalable.

Agent-based Implementation In contrast to RPC, agent-based computing achieves greater customizability by delegating communication to a mobile program defined as the *agent*. The agent can move to an agent execution environment on another machine, known as the *host*, and execute owner-specific code on its owner's behalf. The agent program can be a book search, a filter, or even just a better interface. The crux is that the agent's owner has full control over the powers of the agent, and hence, can customize any transaction. The cost of such customization is the development of a general agent architecture, where research has already provided several examples [18, 20, 15, 26, 31].

Consequently, with the agent as an intermediary for each transaction, the Internet can be perceived and used through an agent, providing a programmable, and hence, customizable interface for its owner.

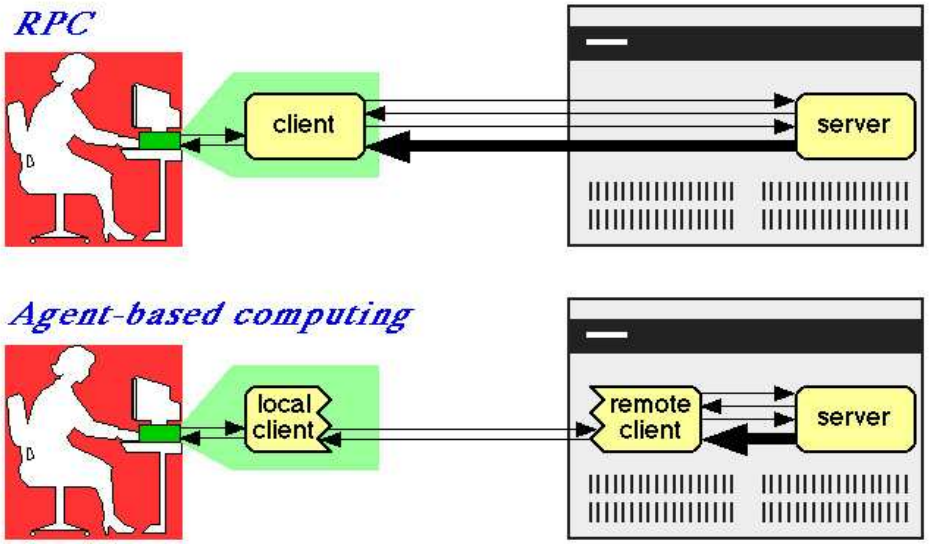

Figure 1: Traditional (RPC) implementation vs. Agent implementation

Similar constructions can be made for each of the NGNAs described in the previous section. In all cases, the traditional method creates problems for the general deployment of each application. It must be noted while agents are never superior to a specialized, traditional implementation of a service (such as the server deliberately providing the client's search method), the primary advantage of agents lies in their flexibility: an agent-based infrastructure can provide the advantages of performance, reliability, and security in a scalable, customizable manner.

1.2.2 Panacea or Prestidigitation?

Agent-based computing is not a new idea, yet there have been few viable, commercially successful applications based upon its principles. A skeptic may point to the lack of applications as a proof of the impracticality of agent-based computing. However, many, if not all, of the NGNAs described are strong, practical ideas with implementation problems addressed by agents.

This paper describes a method for bridging the conceptual gap between agent systems and their applications: namely, agent usage patterns. These patterns are designed to capitalize on the performance, reliability, and security advantages of agents. Through application of these agent usage patterns,

we can build next-generation applications that scale these advantages.

1.2.3 Organization

The rest of this paper is organized as follows. Section 2 details the theory and operation of agents. Section 3 surveys the work done by others in agent systems up to this point and identify the problems behind their approach. A complete listing, description, and analysis of the patterns is in Section 4. Preliminary experiments using these patterns will be described in Section 5 and conclusions will be presented in Section 6.

2 Agent-based Computing

Agents offer flexibility, performance, reliability, and security advantages. Of these agent advantages, the most important is the first, flexibility, as most of the other benefits can be derived from dedicated services.

2.1 Agent Flexibility

The hallmark of agent-based computing lies in its flexibility. As mentioned before, any dedicated client/server protocol for a specific solution will be superior, performance-wise, than an agent-based solution except in the most extreme cases (such as the space probe example given earlier). However, as a general platform for non-dedicated applications that require latency-reducing performance, reliability, and security, agents are hard to beat. Any protocol can be supported easily by writing an agent that implements it and have the application call that agent instead.

Adaptability The flexibility of an agent-based design lends itself naturally to adaptive applications such as multimedia filters [27]. Agents make it easy for such an application to adapt to an environment by encapsulating the necessary procedures for identifying and reacting to changes. Taking the filter example, this means if a multimedia server is sending out high-bandwidth data streams to the agent (which is in turn sending the stream to the client) and the client's connection suddenly experiences a vast loss of bandwidth — frequent in mobile communications — the agent can adapt to this loss by stripping the multimedia stream of characteristics that the client considers less important. When combined with the ability to get close to the source of problems (improving reaction time through the reduction of latency), agents provide an efficient, robust method of adaptation, which is especially critical for mobile applications.

2.2 Other Agent Advantages

In order to justify the construction of an agent architecture for improved flexibility, it is helpful to realize that agents offer several extrinsic advantages: performance, reliability, and security. Depending on their owner's preferences, agent applications can choose to exploit one or more of these advantages.

2.2.1 Performance

Agents offer significant performance benefits, albeit with some cost. Since they have the ability to migrate, they can short-circuit long network distances, thus reducing latency and bandwidth consumed. They can also execute in an environment with significantly more resources than their home machine. However, this comes at the (one-time) cost of establishing the agent system to support it.

Resource Flexibility The simplest and easiest way to improve performance with an agent is to use the movement powers of the agent to move to a machine with vastly superior resources, such as CPU, memory, bandwidth, etc. This also ties in with the inherent flexibility of agents — no longer is the user bound to a single machine's limitations.

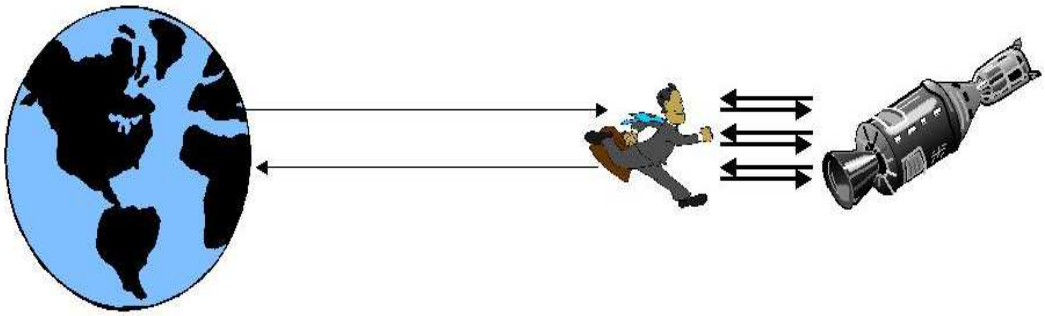

Figure 2: Agent monitoring a space probe far away from the client

Latency Reduction Under the client/server model, messages sent between the client and the server incur a delay for each transmission. If many messages are sent, the cost can be prohibitive. Agents reduce much of this cost by moving one side closer to the other. In the most extreme case, agents can even merge the two sides onto the same machine by having one side send an agent directly onto the other side's machine, reducing the number of network messages to a constant factor: the initial and final movement. For applications that normally require many network messages, performance is greatly improved.

This performance advantage is more important than it seems because, according to current scientific principles, latency is the one factor that cannot be improved upon: it is bounded by the speed of light. While current applications will probably see low latencies due to the bound on distance imposed by intra-planet applications, the potential for enormous latencies in the future (e.g., inter-planet) cannot be ignored.

Bandwidth Reduction An agent can also reduce the amount of bandwidth consumed by a network application by having the agent act as a filter. For example, an agent for a network computer incapable of displaying color can strip incoming multimedia streams of their chrominance factors before being sent to the client. The chrominance would be wasted on the client's computer, while the bandwidth saved could be critical for a client lacking abundant network resources.

2.2.2 Reliability

An agent's ability to encapsulate communication increases transaction reliability: the system is reduced to taking care of a single agent program that can contain specific recovery information. Concrete advantages of this are:

Dynamic Rerouting Agents, being mobile code, can readily recompute an alternative destination if its destination is unavailable. These destinations can be either statically provided by the programmer or dynamically computed through a server that specializes in providing alternative destinations. While traditional methods can also do this, agents improve the modularity of design by encapsulating the movement handling code into the mobile code itself.

Decoupling Computation The ability of agents to migrate to a remote machine allows the agent to decouple their computation from the owner. If the owner's machine is known to be unreliable, either from system or network problems, the agent can be customized in advance to migrate to a more stable environment. This improves the reliability of the transaction by bypassing as much of the unreliable portions of the communication transport as possible.

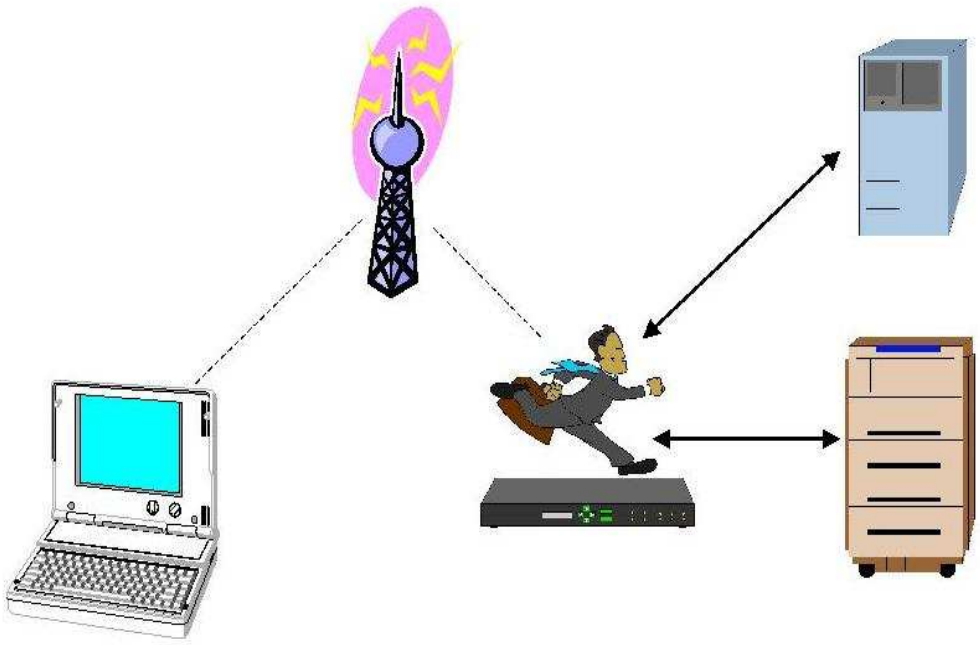

Figure 3: Decoupling computation to bypass an unreliable link

Data Recovery The results of an agent computation can be stored by the agent if its programmer and the agent infrastructure has allowed for it. If the infrastructure contains general agent recovery mechanisms, then the agent can encapsulate all process-specific data recovery mechanisms since its general recovery is assured. The advantage of this setup is that the owner is freed from detailed recovery concerns: the only thing it needs to know is how to reestablish communications with a mobile agent. In addition, the host is not required to keep transaction records for each particular agent, since the pertinent data can be stored within the agent itself. By encapsulating the process-specific recovery data within the critical section of code, the agent, the reliability of the transaction is improved.

2.2.3 Security

Security has a number of different aspects, three of which are privacy, integrity and authentication. As applied to communication, privacy is the ability to keep transaction data private from outsiders, integrity maintains

that the data will be uncorrupted, and authentication assures that both sides are who they say they are. Agents provide a concrete privacy advantage, and the implementation of their infrastructure can help with the other two.

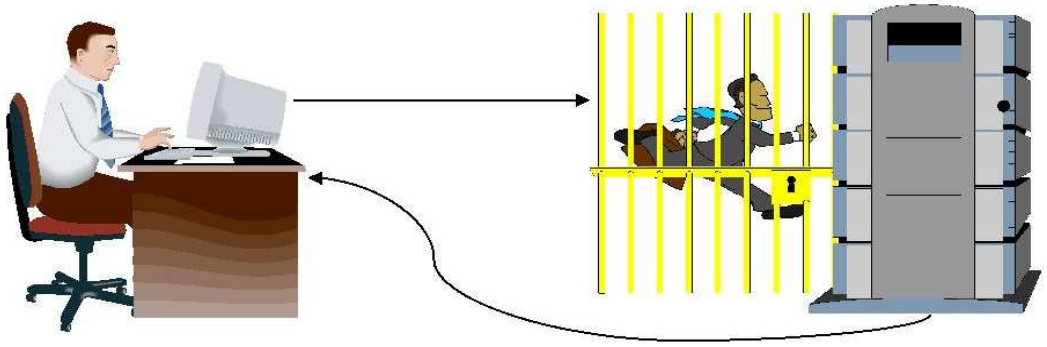

Figure 4: Customizable search controlled by host to ensure privacy

Privacy Agents allow servers to guarantee data privacy while providing clients with customizable access to the data. A visiting agent is under the complete control of its host (presumably the server) and can only carry away a limited amount of data, determined by the host. In fact, the host can completely imprison the visiting agent, obtain the results of the agent's computation, and construct its own trusted agent to send the results back. While a malicious agent may still find a covert channel to transmit data, this approach will greatly reduce the amount of information that can be leaked.

In contrast, traditional servers cannot provide customizable access to its data without depending on the goodwill of its client and the privacy of the network to maintain the value of its information. Given the ease of duplicating digital media, it is impractical to provide such a service and expect to maintain data privacy.

Integrity While there is work on rendering the agent tamper-proof from malicious hosts [38], by and large research has focused on protecting hosts from malicious agents. With the problem of viruses, opening a system to foreign programs without adequate protection measures would lead to unimaginable problems. However, many groups have invented clever methods to guarantee host integrity [9, 21, 16, 17, 32], detailed in the following section.

All of them rely to some extent on the underlying architecture (e.g., by running a "sandboxed" agent interpreter — that is, placing an interpreter in a isolated sandbox where nothing can get out). Integrity is therefore not an intrinsic feature of agents, but a feature of good agent systems.

Authentication Agents act on behalf of their owner, so it seems intuitive to store digital signatures and other authentication mechanisms within the agent. While this seems to be mostly a cosmetic advantage, it does clarify programming by allowing the agent module to handle everything related with the communication. However, in practice, headers can be forged and signatures might be faked. A good agent system will provide a naming hierarchy that will solve some of these problems, but most hosts should not depend on the agents themselves to provide authentication.

2.3 Scalability of Advantages

An important side benefit of the advantages described in the previous section is that they are scalable: as greater demands are placed upon an agent-based system, performance, reliability, and privacy either improve or at worst, are maintained. For example, take the performance advantages. Agent-based communications can reduce latency and bandwidth consumed to a constant factor. Under traditional client/server communications, as the number of messages between the client and server increase, the added network costs begin to severely impair performance. However, an agent can simply migrate to the server and, in the best case, avoid all network costs except for the initial migration and final return, by having most of the communications take place intra-machine at the server. While the agent may need to communicate with the client from time to time, sophisticated agents can minimize this to a small fraction of the former number. Thus agents scale well with regard to the number of messages sent.

The scalability argument can also be extended to reliability. Borrowing from classic fault-tolerance techniques, agents can better overcome intermittent failures by spawning multiple agents to perform identical tasks: as long as a required subset of agents executes successfully, the transaction is carried out[28].

The data privacy that agents provide is scalable. As the number of machines grows, they can be subdivided into domains where each machine only

trusts other machines in the same domain. Confidential inter-domain transactions would require an agent to be sent, with the previously described limits on privacy. As inter-domain traffic escalates, more agents would be sent, but privacy would not be compromised to a greater extent; the privacy exists at a one-to-one communication level, versus a one-to-many. This is in contrast to current methods, where sending data over a network for customized processing has an increased chance of compromised data as the size of the network increases.

Finally, agent integrity can be scalable. As an agent visits more machines, the chance of a malicious host tampering with the agent increases. One can therefore scale agent integrity by increasing or decreasing the number of machines the agent is expected to visit.

2.4 Ramifications of agents

Like all new paradigms, agents create new areas and problems along with their solutions.

2.4.1 Infrastructure

Some of the next-generation network applications would greatly benefit from an existing agent infrastructure: docking bays for retrieving lost agents, metadata servers providing directory information, etc. The lower layers of such an infrastructure would need to be revamped, either through placing agent environments within the network layer (the active network concept), or within a "middle" layer above the network but below the application level (the middleware concept). Research is extensive in both fields, with Active IP [36] and SwitchWare [29] being two active network solutions and CORBA[35] and DCE [3] being two middleware solutions.

2.4.2 Security Problems

While agents provide privacy with flexibility, both agent and host integrity may be severely compromised.

Agent integrity As mentioned previously, agent integrity can be compromised by having a malicious host brainwash an agent into performing an undesirable action. For now, designers of agent-based applications must cater

to the problem of brainwashing by restricting the powers of agents — and correspondingly, the agent's flexibility — or by trusting the invisible hand of the market to weed out the malicious hosts. Neither option is particularly attractive to consumers.

Host integrity The situation is almost just as bad from the other side. Hosts have to guard against malicious agents who try to crack through the safeguards of the agent environment in order to siphon away resources or information. This is not as likely to be as big of a problem as agent integrity since the tendency of recent programming languages (such as Java) towards strong typing and built-in sandboxing techniques offers a strong line of security, but the consequences of an unforeseen loophole would be catastrophic in a global agent infrastructure.

2.5 Applying Agent Technology

It is easy to apply agents to the NGNA scenarios described in the previous section, and see how they solve the problems facing their adoption. Here are a few examples:

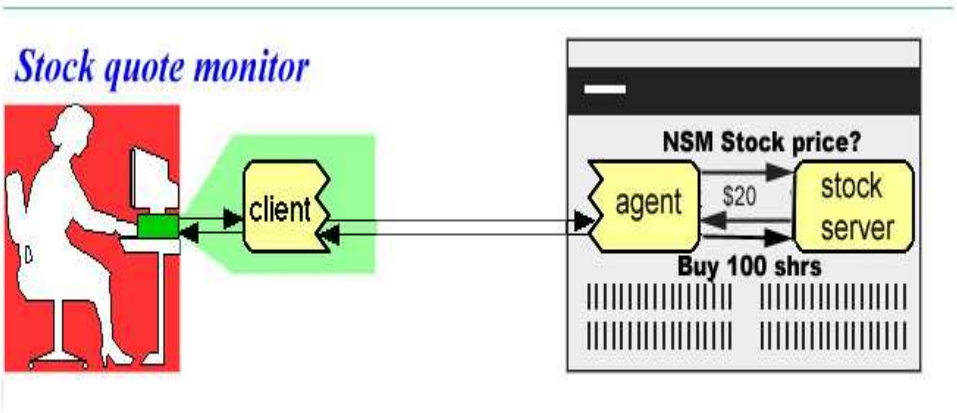

Figure 5: Agent monitoring stock quotes

Agent as real-time monitor An agent is an excellent real-time monitor. It can migrate as close as it desires to the monitored resource, minimizing network traffic and latency. Because of its proximity and programmability, it can rapidly take actions triggered by specified environment changes. Thus, agents are able to double as both the monitors and the decision makers.

To clarify, take the example of stock quotes mentioned earlier. An agent monitoring stock quotes can buy or sell shares at the market price without an intervening latency delay. The client does not need to worry about network delays invalidating transactions. Furthermore, it conserves its own resources by not actively polling the site for price changes. Instead, the agent uses the server's resources (which are presumably more abundant than the client's).

Agents with flexible databases With agents, a client can parse the entire database at the server by using the specialized code that performs the search as the client's agent. On the other hand, the server can guarantee that only results are transmitted back with the agent. This approach maintains the privacy of the data while allowing the client to access the complete data set for user-specific operations.

Agents in a mobile environment Agents provide the general, reliable, cost-effective mechanism for porting programs to a mobile environment. The improved robustness and fault-tolerance of agents allows the client to ignore the effort of maintaining a reliable connection, as most of the time, the agent will not be in the unreliable portion of the network. In fact, the client can even disconnect from the network and asynchronously restore contact with the agent when convenient. Of course, the network will have to provide housing for "homeless" agents, perhaps by using a docking bay [14].

Agents also facilitate the building of adaptable applications by encapsulating the adaptation code. Such adaptation will react quickly as a result of its ability to closely monitor external stimuli, minimizing latency delays. The uncertain amounts and general paucity of resources in a mobile environment are therefore no longer problems for mobile applications; adaptive modules can be easily added in the form of agents.

A final advantage of using agents for mobile applications is that by executing remotely, agents reduce the amount of computation within the mobile computer itself. This results in a performance advantage whenever the migration costs are cheaper than the difference in computation times.

Agents in network management By having agent protocols subsume current network management protocols, the programmability of agents helps one control resources in a faster, safer, and better fashion. System administrators will no longer be restricted to simple peeks/pokes; for example, they will be able to program an agent to alert the appropriate authorities via e-mail that a certain file-server is down. Protocol overhead is also reduced through having updates encapsulated as agents, as individual machines only need to know a general agent protocol, and not a specific management protocol. Agent managers scale better than SNMP-based solutions, as network resources will be consumed in a distributed, rather than a centralized manner. Finally, the security of such communications would tie in with general agent security, eliminating the need to develop a separate security manager for this application.

3 Related Work

3.1 Survey of existing agent systems

While the idea of mobile agents has been around since General Magic introduced Telescript in 1994[37], agents have yet to be supported directly by operating systems. Previous research in this area has concentrated on the design of middleware for supporting agents, sometimes referred to as *agent systems*. This section provides a brief overview of this research.

Four of the most basic issues in the design of agent systems are: mobility semantics (and its ramifications on implementation), communication methods, agent languages, and security features. In this section, an overview of the general approaches for addressing each of these issues will be provided before delving into the specifics for five surveyed systems : Agent Tcl [15], Mole [31], TACOMA [31], Ara [26], and Aglets Workbench [19]. A more detailed overview of these systems can also be found in [10].

Mobility semantics The most important component of an agent system is its treatment of mobility: what aspects of an agent are mobile? Since an agent is a process that can move from machine to machine, differences in mobility semantics revolve around the aspects of an agent that are captured for remote execution. In the following paragraphs, the taxonomy used for defining mobility is taken from [31].

The simplest type of mobility is code mobility. When the agent is ready to move, the system terminates the agent and sends the source code or script to the destination system, where the agent is restarted from the beginning. Since the system does not need to capture the agent's state (which, consequently, does not have to then be re-established at the remote machine), mobility is greatly simplified, at the cost of restricting agent application design.

Since several of the applications discussed above would benefit from state capture, some agent systems assume a stronger form of mobility called data mobility. Data mobility involves the system capturing the machine-independent data portion of an agent and sending it along with the source code. Upon reconstruction, the agent has access to any results from the previous machine(s), although it still must start over from the beginning—but a clever programmer can use the data to manipulate the control structure to return to the point of movement. Data mobility lends itself to a wider variety of

applications than code mobility, at the cost of performance and, of course, additional complexity.

Finally, a few agent systems have the goal of transparent migration. A program can move itself during its execution via a "move command," and after moving, will pick up execution right where it left off. To do this, the execution state as well as the data and code must also be captured and moved. Transparent migration is difficult to implement due to language incompatibilities and machine-dependencies, but a common execution environment such as the Java Virtual Machine may facilitate such transfers. Transparent migration offers the highest degree of flexibility to the programmer at the expense of portability, performance, and system complexity.

Communication methods Agent systems tend to support a variety of communication methods. Some, like Agent Tcl, support low-level socket operations [15]. As sockets are the building blocks for network communication, such low-level support allows a developer to construct any high-level communication protocols desired. On the other hand, it is convenient to provide several high-level protocols. Even if these are not the most efficient tools available, the loss of efficiency is made up for by the reduced complexity for the programmer.

Most systems support some form of message-passing as the principal means of communication between agents, as the asynchronous, flexible nature of message passing complements the autonomous remote execution of agents. In addition, some systems also support remote procedure calls and connection-based communication streams. Finally, some agent systems offer a means of multi-cast communication, the ability to send a message to a group of agents.

One final consideration of agent communications is addressing. While intra-server communication is present in practically every agent system, inter-server communication is not. The problem lies in the complexity of identifying a remote agent whose location can change without notice. While some systems have global directories or proxies to enable inter-server communication, others consider this feature to be a luxury.

Agent Languages One of the most important design decisions of an agent system is whether agents should be compiled, interpreted, or some hybrid [18]? Compiling agents in a manner similar to programs may seem intuitive

to programmers. However, in a heterogeneous environment like the Internet, sending agents as compiled code is impractical due to lack of portability. Furthermore, the security of the host system would depend on the trustworthiness of the foreign compiler—not a viable option.

One approach to these problems is to express agents as scripts that get interpreted on-the-fly. Scripting allows the agent programs to be expressed as text, and interpreting allows an agent system to rely on the home-compiled interpreter to safeguard the environment. Unfortunately, these benefits often come at the cost of performance.

Another approach is to assume that agents will be written in the Java language, resulting in a hybrid solution that achieves acceptable performance with security and portability. In Java, the Java compiler creates inherently portable "bytecodes", which other Java Virtual Machines are guaranteed to interpret. Such bytecodes can be easily checked and monitored for violations of host integrity through Java's strict typing and the Java Security Manager. Further, interpreting bytecodes often leads to faster execution than interpreting scripts since some of the work has already been done by the Java compiler. Consequently, some of the newer agent systems employ Java bytecodes as a portable, secure method for defining agents.

Security Features There are at least three areas of security that must be treated by mobile agent systems: accounting of resources, assurance of integrity, and tools for authorization. Accounting of resources relates to the capability of a system to specify, monitor, and enforce the resources available to agents. Assurance of integrity is defined as protection of both the host's data and its guest agents from other malicious agents (usually through some form of sandboxing). An even more difficult problem is protecting an agent from a malicious host (difficult because the agent often relies on the host to provide its execution environment). Finally, authorization tools allow a system to verify that the owner of an agent is the user whom the agent claims to represent. All of these are highly desirable features that should be a part of an agent infrastructure.

3.1.1 D'Agents

D'Agents, formerly Agent Tcl, is an agent system developed at Dartmouth [15]. The original version, Agent Tcl v1.1, used the Tcl scripting language for agents; all agents were written in Tcl and interpreted by an agent server

running on each participating machine in the Agent Tcl network. The current version of D'Agents supports other languages such as Java 1.1 and Scheme, with support for Java 2 in the works.

D'Agents achieves mobility through transparent migration via the *agent-jump* command. Upon interpreting the command to *agent-jump* to a remote machine, the host saves the execution state and transmits the script and state to the remote machine, whereupon the agent restarts right after the *agent-jump* command.

D'Agents has a flexible agent communication structure by offering several types of communication: sockets, message-passing, and connections. By offering support for low-level socket operations, D'Agents gives resourceful programmers the ability to create a specialized agent protocol. In addition to standard message-passing, D'Agents also allows agents to establish direct connections with each other to provide a dedicated message stream for agents that frequently communicate with each other. Finally, D'Agents allows communication between agents on different machines through the use of a global naming hierarchy to identify each agent in a unique fashion.

The D'Agents project offers security measures for protecting machines from malicious agents.[17]. v1.1 uses Safe Tcl, a modified version of Tcl, to provide a measure of resource control and host integrity. The idea is to have a trusted interpreter running in conjunction with an untrusted interpreter. An agent is first interpreted with the untrusted interpreter, which traps to the trusted interpreter whenever a "dangerous" command (i.e., one that may consume too many resources if not checked) is given. The trusted interpreter then evaluates the command with the aid of a resource manager agent to see if the agent is overstepping its bounds. This can be extended to a sandboxing approach for maintaining host integrity, although v1.1 did not support it. For authorization tools, D'Agents supports PGP to digitally sign and encrypt agents for authorization confirmation. Despite these precautions, the system does not address the inverse problem of protecting agents from malicious machines.

3.1.2 Mole

Mole is a Java-based agent system developed at the University of Stuttgart in Germany [31]. All the agents are written in Java and compiled into bytecodes for transport and interpretation.

Since Mole interprets agents using the Java Virtual Machine, it can cap-

italize on the security advantages of Java such as bytecode verification. For access restriction, Mole divides agents into two classes: user (mobile) and system (immobile). Only system agents can access system resources, and since they are immobile, they must be created by the system user (who is assumed to be trustworthy). User agents communicate with system agents in order to get work done, but do not actually interact with the resources themselves.

Mole's migration concept extends only to data mobility. The marginal returns from enabling transparent migration were seen as inadequate for the first version (although future plans include implementing transparent migration). Thus, when a user agent moves to another site, it must restart execution from the beginning, although it has access to previous results.

Finally, Mole offers both message passing and RPC (through Java's RMI facility) protocols for agent communication. While global inter-agent communication is supported, it is not location-transparent: in order to communicate with an agent, one must know the location along with the name of the agent. Location transparency may be supported in a future version.

3.1.3 Aglets Workbench

Aglets are the agents of IBM's commercial Java-based agent system, known as the Aglets Workbench. Aglets are most similar to the Mole system, as they both use the Java Virtual Machine and bytecodes as the foundation for their agent framework. Thus, they share many of the same security and language attributes. However, the two do differ in a few aspects.

First, aglets are able to transparently migrate from host to host. Second, the Aglets Workbench offers an enormous variety of communication methods: synchronous/asynchronous message passing, RPC via RMI, group communication via "whiteboards", priority queues, remote messaging, and more. The addressing is global and transparent; each aglet has a proxy associated with it that keeps track of its location, and all communication with an agent is done through its proxy. Third, there is an Aglet Security Manager that performs additional aglet-specific integrity checks on top of the generic Java Security Manager. Finally, there is no division between the powers of system aglets and user aglets. All aglets have the potential to do the same things; however, aglets created locally tend to be tagged with a "trusted" flag that tells the Aglet Security Manager to be more lenient.

3.1.4 TACOMA (TUX)

TACOMA is another agent system, developed at the University of Tromso in Norway [20]. Like Mole, it eschews the added complexity of transparent migration in favor of the simple data mobility approach. Code and data are stored into TACOMA abstractions called *folders*, which are collected into another abstraction called a *briefcase* and sent to a remote machine. Upon arrival, the host unpacks the folders from each briefcase and executes the code in the CODE folder.

TACOMA supports only local communication between agents. This simplifies communication methods to one type: Agents *meet* with one another, which involves the transfer of a briefcase from one agent to another, similar to a synchronous message passing scheme. Since meetings can only occur between two agents at the same location, agents in TACOMA do not communicate with agents on different hosts–avoiding the addressing problems through limiting powers.

While several agent systems espoused the goal of supporting multiple agent languages, TACOMA was the first to realize it. An agent's briefcase specifies the type of code it is carrying. Consequently, the host knows how to handle the code sent to it, provided that the agent is written in a language it supports. TACOMA v.1.2 supports Tcl, C, C++, ML, Perl, Scheme, Python, and VB, with Java support expected in later versions.

3.1.5 Ara

The Ara system's chief goal is to provide transparent migration while supporting compiled code for performance flexibility. The transparent migration is similar to Agent Tcl's: both provide a *jump*-type of operation that saves the execution, code, and data state of an agent to be restored at the remote site.

Both architectures use Tcl for their main agent scripts, but Ara also supports agents compiled in C. Ara avoids the portability and security problems presented by compiled agents by dividing agents into mobile and non-mobile, as in Mole, and restricting compiled agents to the latter set. Ara also claims to have an extensible architecture for supporting other interpreters, and the developers are working on adding a Java interpreter to the architecture to prove their point.

Ara allocates resources through an allowance system, determined when

an agent moves. Ara has defined an interface to the system through which agents can inquire about and trade allowances to other agents. The agents are also protected from each other through an address-space type of security: agents can only affect what is in their "space" and the core/interpreter is the controlling entity that can interact between spaces. While the core is not directly accessible to agents, its functions can be called through the use of stubs.

Finally, Ara uses the concept of a *service point* to handle inter-agent communications. Service points are essentially meeting places created by agents so that they can transmit data between each other in a synchronous, blocking fashion. A special feature of service points is that more than one agent can "meet" at a service point, providing multicast services. Since service points are "located" at a host, the abstraction neatly sidesteps the addressing problem by forcing all agents to come to the same place.

3.2 Summary

This section has covered some of the more important features and details of several existing agent systems. It is noteworthy that there has been significant prior research in the area of agent-based systems. These systems have done an effective job of supporting agents, and mechanisms for supporting agents are becoming well-understood. We now describe the patterns we have identified that allow these systems to support scalable NGNAs.

4 Agent Usage Patterns

An agent usage pattern describes the use of an agent in a fashion that capitalizes on the advantages of the paradigm, in terms of flexibility, performance, reliability, or security. The defining attributes of patterns will be described first, before specific patterns are identified and analyzed.

4.1 Pattern Components

Patterns consist of *machines*, *objects*, and *channels*.

Machines A machine represents the system hardware. All agent patterns contain a client machine (the machine of the client/user) and a host machine (a machine that executes agents). A data server machine, which is analogous to the server machine in the client/server model, is not required, but is a frequent component in patterns that seek to be compatible with existing network services. Also, the host machine is not required to be a distinct third-party machine; either the client machine or the data server machine may double as the host machine.

Objects Objects represent running software, and are divided into three categories: agents, clients, and servers. All agent patterns require at least one agent object (an agent) and at least one client object (the program to which the agent is responsible). A server object is frequently used, for the same reason — compatibility with existing services — as server machines above.

Channels Channels represent the communications between objects and are categorized by the Communication attribute (described in detail below). This attribute stipulates the allowed types of communications in a pattern.

4.2 Pattern Attributes

There are three basic attributes for patterns: location, communication, and movement. These attributes heavily influence its performance, reliability, and/or security. Delineating each pattern into these attributes makes it easier for a designer to analyze a pattern's advantages and disadvantages.

4.2.1 Location

The first attribute of a pattern is the identity of the host machine, and hence, the location of agent execution. The host machine can be the client machine, the data server machine, or an independent third-party. This attribute is the primary factor behind the analysis of a pattern's performance. However, it also influences reliability and security. Here is a breakdown of what each location gives the application:

Agent executes on data server machine In this case, the data server machine also runs an agent server. Thus, both the speed of computation and the reliability of the agent is equivalent to a normal program running on the data server. Furthermore, any latency costs or instability from the communications between the agent and server objects can be ignored. The privacy of the server data can be enforced if the server denies the agent a communication channel back to the client. Finally, the client receives no guarantees about agent safety or server fairness: the agent is entirely at the mercy of the server.

Agent executes on client machine Here, the client runs the agent server, and the agent never physically migrates away from the client. Consequently, the speed and reliability of the agent is equivalent to a normal program running on the client. Any problems with the communication channels between agent and client objects, such as unstable connections in a mobile environment scenario, are completely eliminated. This essentially emulates traditional client/server applications, with the added bonus of agent flexibility. Furthermore, data privacy is voided, since the server is forced to transmit the data to the client machine, but the client gains the assurance of agent integrity: the agent cannot be tampered with if it never leaves the client.

Agent executes on third-party machine In this scenario, the agent does not rely on the ability of a traditional client or server to support its execution; rather, it moves to a third-party machine that runs an agent system. This way, the agent performs at the speed and reliability of the third-party machine. While it is impossible to completely eliminate performance or reliability problems from communications, if communication between one side and the agent is minimal, then there will be a corresponding benefit. For example, in the Bypasser pattern, the

client sends the agent to a neutral server and then only communicates with the agent at the end of execution, thus reducing latency costs and bypassing a potentially troublesome link.

A disadvantage of this approach is that it reduces privacy, as now two outside machines get access to the data. However, a neutral host has the unique advantage of providing a fair transaction model: both the data server and the data consumer can send agents to the same neutral host and experience an equivalent level of trust. Of course, the integrity of both agents is subject to the integrity of the neutral host.

4.2.2 Communication

As with location, there are three basic choices for communication between objects. The choices are : no communication, simple communication (equivalent to message passing between objects), and dedicated communication (equivalent to a channel that is opened and maintained by the objects).

No object communication Since there is no communication, any potential communication problems, such as limited bandwidth between the objects, can obviously be ignored. Also, the privacy of data is assured, because no communication takes place.

Simple communication If a pattern describes communication as simple, the communication is not expected to be a performance bottleneck, so the latency cost of such communications is negligible and should usually be ignored for analytical purposes (exception : when the cost of a single transmission is significant, such a client sending a message to a very distant agent). However, the reliability of the communication is now based upon the reliability of the weakest link between the two objects, albeit slightly since the communication is not expected to be maintained for long periods of time.

The privacy of server data is dependent on whether foreign objects can communicate beyond the boundary of the data server machine. If so, the pattern does not guarantee data privacy.

Dedicated communication A pattern with dedicated communication between two objects has both its performance and reliability dependent on

the weakest link between the two objects. Again, as with simple communication, privacy cannot be guaranteed if communication generated from a foreign object leaves the confines of the data server machine.

4.2.3 Movement

The migrational behavior of the agent — its movement — is the final attribute of a pattern. A pattern can choose an agent to act as a one-shot (moves at most once), a boomerang (moves to remote site, then returns, as a thrown boomerang), itinerant (moves according to a predefined itinerary), or autonomous (moves as its execution dictates).

One-shot A one-shot agent guarantees privacy provided all the other privacy factors (no communication outside data server machine, execution at data server) are met. It is also simpler to model and implement than the other migration models, and is the assumed movement model for all patterns unless otherwise defined.

Boomerang An agent using the boomerang movement can be analyzed exactly as a one-shot, except that the privacy guarantee is voided and it is slightly more complex to implement. The reason for the added complexity is that the underlying agent system must provide a mechanism for recovering agents from the network, in the case of a client crash. Since this mechanism must be in place for this model to function correctly, a pattern using a boomerang over a one-shot has a stronger reliability model in terms of receiving agent results.

Itinerant The itinerant agent's advantage is the ability to visit multiple machines in a predetermined order. While the one-shot model does not prevent this (each one-shot agent can spawn a child agent that goes to the next destination), the advantages of the itinerant approach over the one-shot are twofold. First, it is more intuitive and simpler to program an itinerant agent given direct support for this model: all of the destinations are in a simple list and there is no need to have a separate branch for each stage of movement. Second, the encapsulation of all vital data into a single agent makes it easier to program crash recovery. There is no need to identify the correct computer from which the agent needs to be restarted, as the itinerary is stored within the agent and the agent can restart on any machine in the itinerary and

maintain "travel state". In addition, no connections need to be restored to a parent agent which may be awaiting results.

Autonomous As of yet, while no patterns require an autonomous movement model, it is included for completeness. The autonomous model provides nothing at all in terms of reliability, security, or performance, but is the most flexible movement model for applications, as there is only one restriction on movement: the destination must be running an agent server. This model supports the ability of an agent to move to a remote site, and make a decision on-the-fly to move to a different site. It is the traditional movement model for mobile agents.

4.3 Scalability Analysis

Through recognizing a pattern's fundamental attributes, a designer or user can easily scale agent advantages by choosing a different attribute within the same category.

Scaling through location All agent advantages can be scaled through location, as the performance, reliability, and security of the execution environment strongly influence the performance, reliability, and security of the agent. The user can easily customize these advantages by choosing the appropriate machine: i.e., a powerful, but untrusted host for performance over security, or a stable, safe, and highly congested host for reliability and security over performance.

In addition, the location attribute even enables users to scale performance in a network of machines relatively equal in power, as the choice of an agent execution location is the overriding factor behind the calculation of latency costs of network messages. The designer, knowing which communications are dedicated, can suggest certain locations as optimal for minimizing these costs, but the user can customize the quality of service by selecting another location to attain other advantages (such as reliability or security).

Scaling through communication Changing the communication attributes of a pattern influences reliability to a large extent. The user can determine which communications should be minimized, as there may be congested or troubled links to avoid. Thus, in an unreliable environment, the user can choose a pattern that minimizes the use of troubled links by reducing the

amount of communication over the links. For example, while dedicated communication is usually the communication of choice between objects, if an application has trouble with the link between the objects and knows that the amount of data to be sent does not require maintaining a channel, the designer should scale down the pattern's communication to simple, message-based communication.

Scaling through movement The movement attribute primarily influences security, as it determines the amount of foreign exposure an agent encounters. In the case of the one-shot, the agent only moves to the host machine and never leaves, providing minimum exposure to both the agent and to the server's data. A pattern using the boomerang movement model would face the same amount of exposure, but would be unable to guarantee privacy of server data. The itinerant and autonomous models offer greater flexibility in mobility with a correspondingly greater risk in agent integrity, as the more machines an agent is exposed to, the more likely it is that an agent will fall victim to a malicious host.

Reliability is also scalable through changing the movement. The boomerang model guarantees a return of the agent, while the itinerant model enables a simpler crash-recovery scheme.

4.4 Agent Pattern Catalog

We now present a catalog of the patterns identified up to this point. For mnemonic purposes, the patterns have been given names which reflect their usage: e.g., Monitor, Commuter, Isolator. A template similar to the one used in Erich Gamma's Design Patterns [12] will be in use in the description of each of the following patterns. The template will consist of the following fields:

Description How the pattern works, and its intent.

Key Application The next-generation network application(s) that would use this pattern.

Structure A graphical model of the pattern, using the following notation:

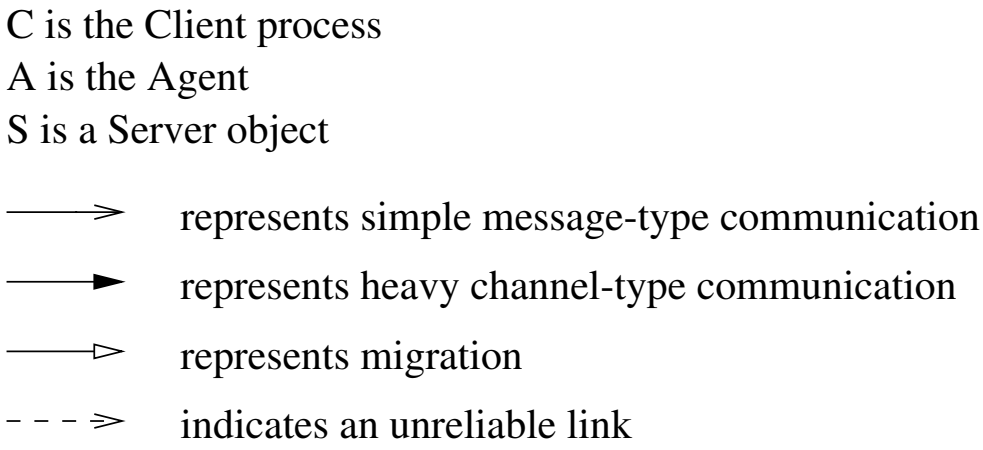

Figure 6: Structure graph key

4.4.1 Analysis

Describes the pattern's advantages and disadvantages.

4.4.2 Bypasser

Description Many applications require frequent communication between client and server. An unstable network or client can destroy the utility of such applications. The Bypasser is an agent that will help these applications by being used for the purpose of circumventing an unreliable link. The agent is sent to transact on behalf of the client. The agent need not be located at the same site of the server. It is sufficient to have the agent go far enough towards the server so that any unreliable network links will be bypassed; or, in the case of an unstable client, to have the agent just get off the client. Upon completion, the agent returns to the client, invoking any recovery methods built into the system in case of failure.

Key Application The Bypasser is oriented towards any non-interactive application on a mobile platform that needs to communicate frequently with

a more reliable server. The mobile computer uses the section of code that communicates with the server as the Bypasser.

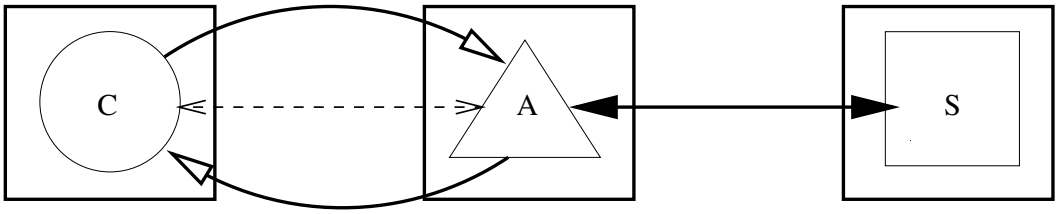

Figure 7: Structure of the Bypasser pattern

Analysis The Bypasser's primary advantage is reliability with performance. The use of the unstable link is minimized. Since the pattern uses the boomerang movement model, which requires an agent system with recovery mechanisms, a return is guaranteed despite being made over the unstable link. While traditional networks can provide similar recovery mechanisms, such recovery is not guaranteed, nor will the average extra overhead needed for each traversal of the problem link be efficient.

The Bypasser also gives a slight performance bonus by reducing the latency involved in messages sent between client and server, but this is not the driving feature of the pattern.

The Bypasser should not be used in an interactive application such as a web browser, as this defeats its purpose of bypassing the client's unreliable situation.

4.4.3 Commuter

Description The Commuter is a program that is sent to execute on another machine in order to capitalize on the remote location's (server's) resources.

Key Application The pattern is the agent model for implementing resource sharing over remote networks.

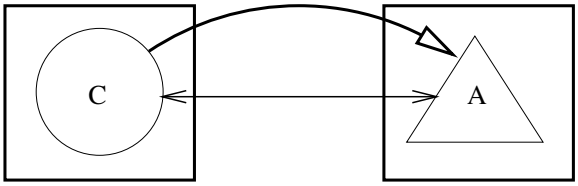

Figure 8: Structure of the Commuter pattern

Analysis This is primarily a performance or reliability-boosting pattern with a touch of security. By giving the client choice of execution environment, the client can scale performance vs. reliability by picking the appropriate environment. In addition, the agent system infrastructure should provide built-in security and resource control procedures to prevent foreign programs from abusing the resources provided to them. Since inter-machine communication tends to be more expensive than intra-machine communication, this pattern should be primarily used for non-interactive, computationally-intensive processes whose home environment lacks either the necessary or sufficient resources to execute.

4.4.4 Isolator

Description The Isolator pattern forces the agent to execute on the server, while denying it the ability to communicate with the client. Upon completion, the agent sends its results to the server. The server, being the host, then destroys the agent and sends the results to the client.

Key Application Any application which needs to safeguard privacy of data without restricting flexibility, such as the flexible database, will want to use the Isolator. In the case of the database, the database only forces agents to go through the Isolator pattern : it receives agents, permits them to execute and achieve a result without outside communication, destroys the agent, and relays the result back to the client.

Analysis This is a pattern for privacy. By restricting the movement and communication of the agent, the server/host can bound the amount of information (in bits) sent to the client, and thus guarantee a certain level of

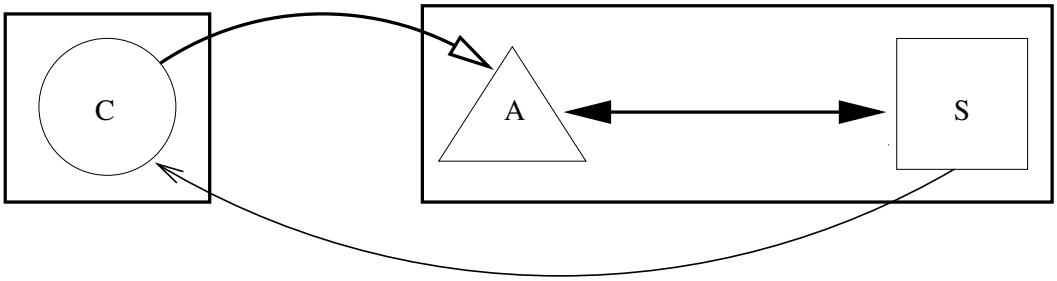

Figure 9: Structure of the Isolator pattern

secrecy of its data. The pattern may provide other benefits (such as eliminating problems related to network communications between agent and server), but these are peripheral to the privacy advantage of this pattern.

4.4.5 Monitor

Description As a Monitor, the agent migrates to a remote location that is closer than the client to an object that needs to be constantly monitored. Ideally, this would be the machine upon which the object is located, but external circumstances may force the agent to monitor from a nearby location instead. The agent is given the identity of the object to be monitored, trigger procedure(s) which are executed upon the object reaching a state of interest, and the states of interest to the client.

The agent's normal procedure is to check the state of the object every so often (the time between checks being set by the client). If the object reaches an interesting state, the appropriate trigger procedure is executed.

The agent may return a result to the client.

Key Application The quintessential application for this pattern is real-time monitoring of a remote object. Take the stock quote monitoring example from the introduction: an agent could monitor a stock price, and when the price hits a certain amount, it executes a transaction on behalf of the client. Other applications which would find this pattern useful include the multimedia filter (monitoring state of network to adapt), resource sharing (monitoring remote workstations for idleness), and distributed network management (monitoring systems for problems).

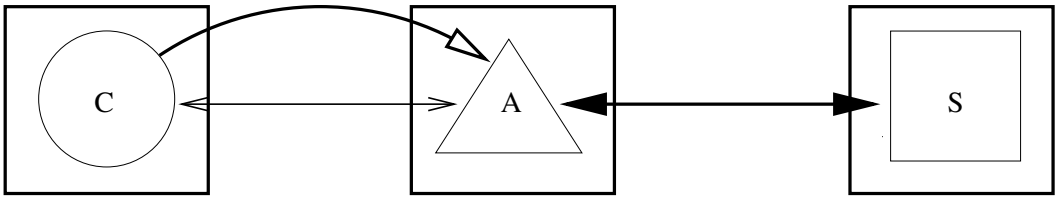

Figure 10: Structure of the Monitor pattern

Analysis The Monitor agent pattern clearly reduces the latency costs of monitoring by reducing the distance traveled by most of the traffic (the monitoring requests and responses). In the special case of having the Monitor agent at the site of the object, the Monitor completely eliminates the network latency during monitoring. There is still a latency cost for initial and final transmission, but as the nature of monitoring requires many messages to be sent, there is a scalable performance gain from this special case(the number of network messages is constant despite the number of monitoring attempts).

Furthermore, the Monitor is more reliable than traditional monitors. It uses at least a portion of the connecting network between the client and the object less frequently, and is correspondingly less susceptible to network failure.

4.4.6 Interface

Description The Interface is an agent that acts as an intermediary between the client and a server. There is no restriction on agent location: the agent can execute anywhere and still perform the function of an Interface. All communication between client and server is expected to go through the agent, which can perform customized operations upon the data before sending it onwards.

Key Application The Interface should be used for interactive applications that need customization without requiring much effort on either side. The NGNA corresponding to this pattern is the electronic marketplace. As a buyer, a user can have an agent that acts as a flexible, transparent front-end to personalize the interactions with a seller. Also, applications which need to filter complex server data (such as multimedia, or verbose HTTP headers) for

the special needs of the client would also benefit from applying this pattern.

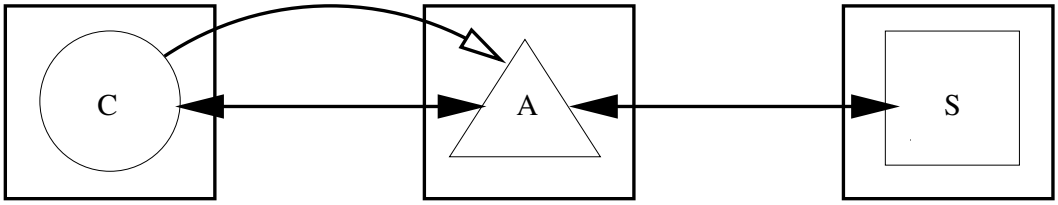

Figure 11: Structure of the Interface pattern

Analysis An Interface interpolates an agent into the communication flow. While the agent can be fine-tuned to improve the application's performance (via data filtering), or reliability (via checkpoint services), the underlying advantage of the Interface is the customizability of established interactive applications, such as web browsers. For example, a client who prefers to read in Swahili can interpose a Swahili-translator agent in front of the browser to display the data in Swahili.

Adding such a specialized interface reduces overall performance through the overhead of receiving and sending data an extra time, but this is a small price to pay for the added customizability.

4.4.7 Rover

Description The Rover models the agents that purposefully visit several sites in sequence. An agent using this pattern is given an itinerary of sites to visit and then visits every site on the list.

Key Application Network management applications and applications using the electronic marketplace could benefit from this pattern. In network management, one could use a roving service agent that continually hops from machine to machine to check for problems. In the electronic marketplace, a client can batch requests into a single agent which can visit several sites before returning results to the client.

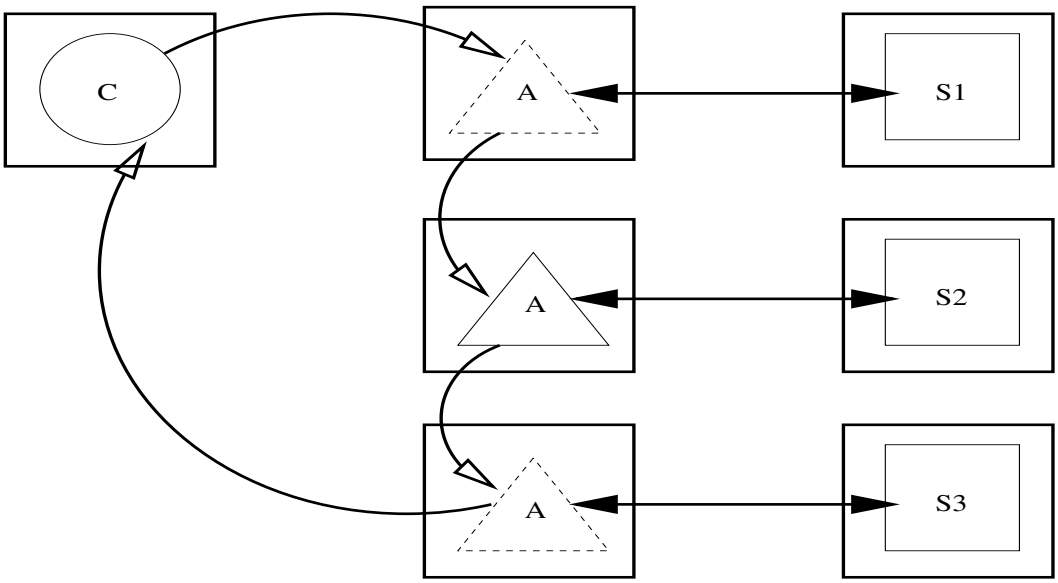

Figure 12: Structure of the Rover pattern

Analysis The Rover pattern is the only pattern cataloged here that uses the itinerant movement model. The advantages of itinerant movement, as described earlier, still apply: intuitive design and a streamlined recovery model.

The Rover may also grant a performance advantage: it may achieve significant reduction in latency costs if the client is far away from the cluster of target servers. However, given the interconnected nature of network topologies, it is unlikely that this will be a strong factor. A more likely performance benefit will be the decongestion of network traffic by distilling the multiple communications of a centralized server into a single mobile agent; but this is a global benefit, not an individual one.

4.5 Summary

Now that the patterns have been introduced, with their attendant analysis, it remains to show how one can use these patterns to develop agent applications. In the next section, we describe some applications that were built using these patterns.

5 Experiments

Having defined the agent usage patterns and analyzed their individual strengths and weaknesses, we present some preliminary experiments that show how these patterns can be used in a conceptually modular fashion to construct successful, scalable agent applications. In the following experiments the goals were to :

- Develop useful applications based on agents.
- Show that patterns facilitate application development.
- Scale agent advantages with these patterns.

5.1 Completed Applications

As a foundation, a primitive agent system in Java was developed. Its architecture relies on a specialized, intermediary server to host agents, called the CAS (Client-Agent-Server) Server. Three applications were then developed: the Shepherd, the Trader, and the RoboTrader. The Shepherd is an agent application for resource sharing and is based upon the Commuter agent pattern. The Trader is an application that uses traditional communication methods to monitor a primitive stock server for the right moment to buy shares. The RoboTrader uses code from both the Shepherd and the Trader to provide an improved-performance, agent-based alternative to the Trader.

5.1.1 The Shepherd

The goal of the Shepherd is akin to that of a real shepherd guarding his flock. The Shepherd application is entrusted with a program (henceforth referred to as the Sheep) submitted by the client. The Shepherd is then responsible for finding a place to execute the Sheep. If it thinks the Sheep would be better off running in some other machine's pasture (agent environment), it will send an agent with the Sheep to that environment. During the Sheep's execution, the Shepherd agent operates in another thread, monitoring the environment for any problems and taking care of them if they arise (e.g., if the Sheep is about to run out of resources for execution, it purchases more).

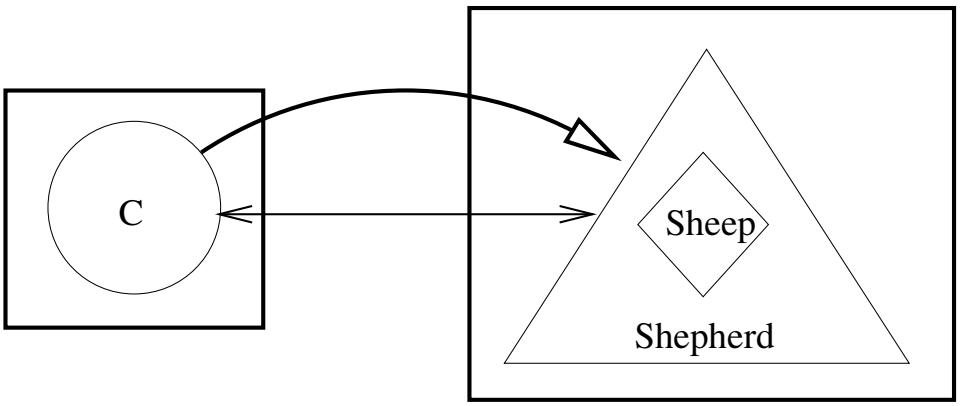

Figure 13: Structure of the Shepherd application

Theoretical Analysis The Shepherd, being a Commuter-based application, aids performance and/or reliability while providing security to the remote server through use of the established agent infrastructure. Both machine performance and reliability are dependent on the machine the Shepherd chooses to execute on, and the relative importance of the advantages is customizable by having the user prioritize them appropriately. Unless the programs being transferred are unusually large and their transfer time is significantly longer than their execution time, network performance and reliability will be not be a factor, as only simple communication channels are used to transfer the Shepherd and Sheep over to their destination. Finally, the Shepherd enables quick adaptation to environment changes by controlling the Sheep on the same machine.

Implementation Details Upon startup, the Shepherd is given the attributes of the Sheep : the compiled Java class file and its arguments. Then, for the purposes of this exercise, the Shepherd arbitrarily decides to move an agent containing the Sheep to another machine, "santorini", upon which a CAS server is running. It contacts the CAS server at santorini, sends an agent, and opens a simple socket-based connection between the agent and the client to send over the Sheep's attributes to the agent. The agent then uses this information to execute the Sheep on the remote site, capturing the output stream in the process. Finally, the agent sends the output stream back to the original machine for display.

In this implementation, the monitoring aspect of the Shepherd was not

fully implemented. The only contact between Shepherd and Sheep was for the Shepherd to capture the Sheep's output stream and send it back to the client upon completion. Also, for this simple experiment, the Shepherd adheres to the one-shot movement model, moving at most once. A more useful Shepherd may be able to improve performance at the expense of security by being able to move the Sheep to different servers.

5.1.2 The Trader

The Trader application trades stocks through more traditional means of communication.

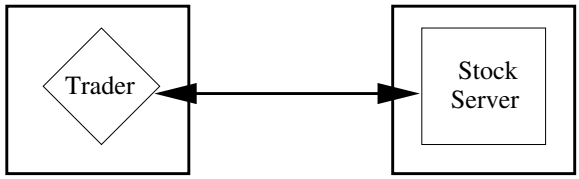

Figure 14: Structure of the Trader application

As one can see from figure 14, the Trader is designed with a straightforward client/server approach. The client establishes and maintains a connection with the server. The server sends messages to the client whenever a stock price changes. The client then chooses to buy or sell shares depending on the price changes.

Theoretical Analysis The Trader's performance is dependent on three factors: the speed of the data server, the speed of the client machine, and network delays. The reliability of the Trader is dependent on the reliability of the data server machine, the client machine, and the network; if any fail, the application fails.

Implementation Details The Trader connects via network sockets to a remote stock server. The server outputs a ticker, showing the changing stock prices. The Trader is programmed to send a buy request when the price of a certain stock falls to a certain value (both chosen by the user as input arguments), and then exits. The server is programmed only to accept the first buy order it receives, and refuses all other buy orders until the price of the stock changes.

5.1.3 The RoboTrader

The RoboTrader has the same goal as the Trader. The difference lies in its implementation: agents are used to scale performance, reliability, and security.

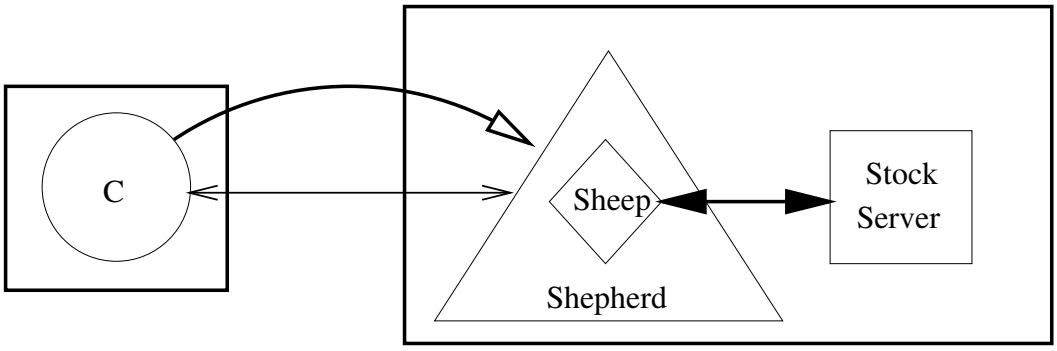

Figure 15: Structure of the RoboTrader application

Theoretical Analysis The RoboTrader's structure is based upon the Monitor pattern. According to the analysis from the catalog in the previous section, the RoboTrader's performance and reliability is based solely on the data server's performance and reliability. The network and client machine only figure into the equation during the initial setup.

In addition, when at the data server, the RoboTrader pays no network latency costs for monitoring the stock. The only costs paid are on startup and finish, when the agent migrates to the server and sends the result (failure or success) back to the client. This is a stark contrast to the Trader, which wastes valuable time by having to pay network costs for each monitoring message. Even if the RoboTrader is forced to use a third-party host as a base for monitoring, performance will always be at least as good as the Trader, because it can reduce latency costs by moving to a closer, intermediary site (unless, of course, the hosts vary widely in performance).

Implementation The RoboTrader application was created by combining the previous two applications: the Trader application was used as the Sheep in the Shepherd application. This method was arrived at through side-by-side comparison of the Commuter and Monitor pattern and figuring out what

needed to be done to transform the former into the latter. The Shepherd attempts to move the Trader Sheep to a machine as close as possible to the stock server, to minimize latency costs; if the stock server itself is unavailable, the Shepherd can adapt by targeting a machine close by, or in the worst case, stay on the client machine. The intent of this design is to provide a flexible performance gain at the expense of slower startup and delayed output to the client (which has to go through the extra Shepherd Agent).

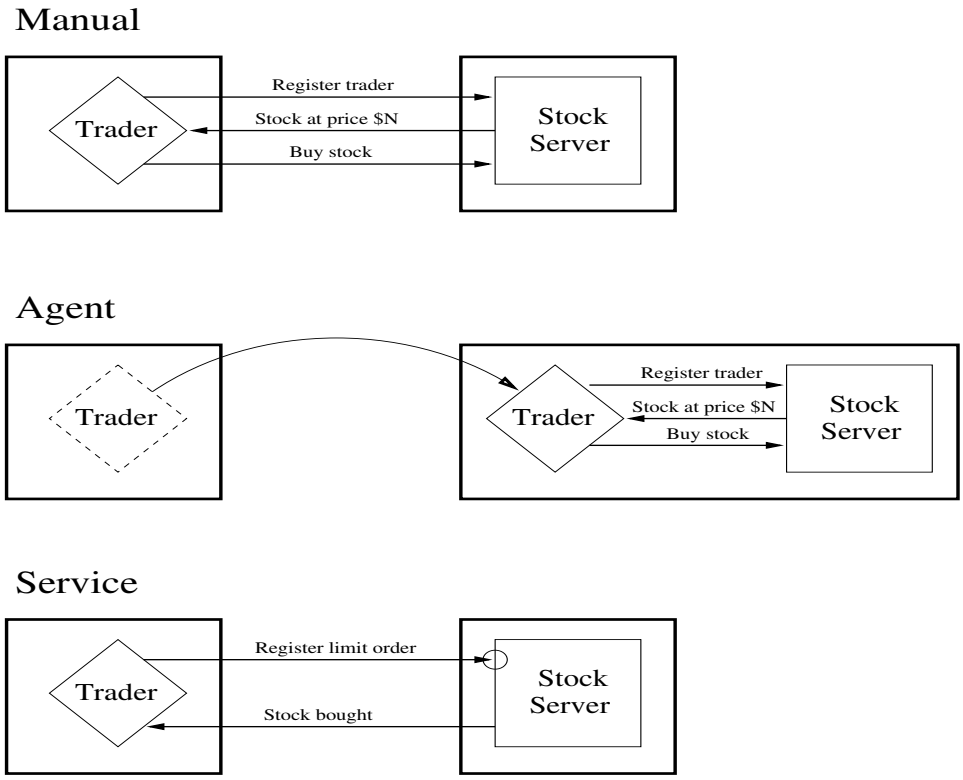

Figure 16: Three Network Paradigms

5.2 Results

5.2.1 Experiment 1: Evaluating Paradigms

The first experiment was to evaluate the relative performance of three network paradigms (shown in Figure 16):

1. traditional RPC – the Manual paradigm

2. customized incorporation of services – the Service paradigm

3. agent-based computing – the Agent paradigm

The Manual and Agent paradigms were defined in the Introduction and correspond to a traditional method of communication with a server, and the RoboTrader application, respectively. The third attempts to realize the customizable advantages of agents without actually using agents. This is done by directly incorporating, or "hard-wiring," the customizations into the primitive server services. This removes the need to support the agent infrastructure, as well as the overhead of agent-server communication, at the cost of flexibility that programmable agents provide.

In the following experiments, the performance metric used was the time it took for the server to receive a "buy" command after the stock hit the target price, essentially a measurement of the network latency time between the decision-maker and the server and the CPU time alloted to the decision-maker.

To introduce latency as an important factor, the client machine, a Sun Ultra 1, was stationed at Carnegie Mellon University in Pittsburgh, approximately 2500 miles away from the server machine, a Sun Ultra 10, in San Diego. The results, after 10000 iterations for each application, are summarized in the following table:

Paradigm	Response Time (ms)			Confidence (95%)	Standard Deviation
	Min	Max	Mean		
Manual	71	3474	79.6	$\pm$ 1.8 ms	91.4 ms
Service	0	11	0.051	$\pm$ 0.005 ms	0.27 ms
Agent	1	14	1.161	$\pm$ 0.008 ms	0.39 ms

Table 1: Comparison of Response Time for Various Paradigms

As can be seen from the confidence intervals, the averages are statistically significant. These results show that, not surprisingly, the best performance occurs if the server can be hard-wired with the action that the client requests. However, in practice, not all servers will be able to anticipate every need of the client. The manual "non-agent" approach for flexibility results in severe performance penalties, and also a highly variable response time, as shown by

the high standard deviation. The high deviation is a result of intermittent network congestion and timeouts.

5.3 Experiment 2: Application-based Comparison of Scalable Performance

As a supplement to the previous experiment, an application-based performance comparison was constructed. In this experiment, the central result depended on whether the stock monitoring application was successful in purchasing its stock at a desired price. A competition between the two stock monitoring applications, the Trader and the RoboTrader, was set up, diagrammed in Figure 17. In each case, the client is the previously mentioned machine at CMU (*federation*), while the stock quote server is the previously mentioned machine at UCSD (ursus).

The first experiment was a control experiment. Both the Trader and the RoboTrader were started on *federation*. Each program competed for the right to purchase a stock on *ursus*. Since both traders were running on the same machine, this control experiment was expected to produce a success rate of approximately 50In 5000 trials, the RoboTrader bought the stock 2538 times to the normal Trader's 2462, giving a sample success rate of 50.8% with a 95% confidence interval of 1.4%.

In the next phase, the RoboTrader migrated to *ursus*, the server, in order to demonstrate the optimal case for agents. After 5000 trials of this new setup, the RoboTrader beat the Trader every time, instead of half the time. The elimination of the network latency from the critical path gave the Trader no chance to beat the RoboTrader.

To test flexible support of scalability, in the last phase the Shepherd aspect of the RoboTrader was given a list of sites to run on, with *ursus* as the primary choice and *abmem*, also a Sun Ultra 10 on *ursus*'s local subnet, as a secondary choice. Then, *ursus* was modified to deny support for agents. The Shepherd aspect of the RoboTrader was able to react to this turn of events by migrating to its alternative machine, *abmem*. From *abmem*, it competed with a Trader on *beowulf* for the same stock at the same price.

Under these conditions, the RoboTrader was able to purchase the stock at a 100% success rate in 5000 trials, a success rate equivalent to the second experiment, where the RoboTrader ran on the stock server *ursus* itself. Thus, the experiment showed that agent-based applications can still achieve out-

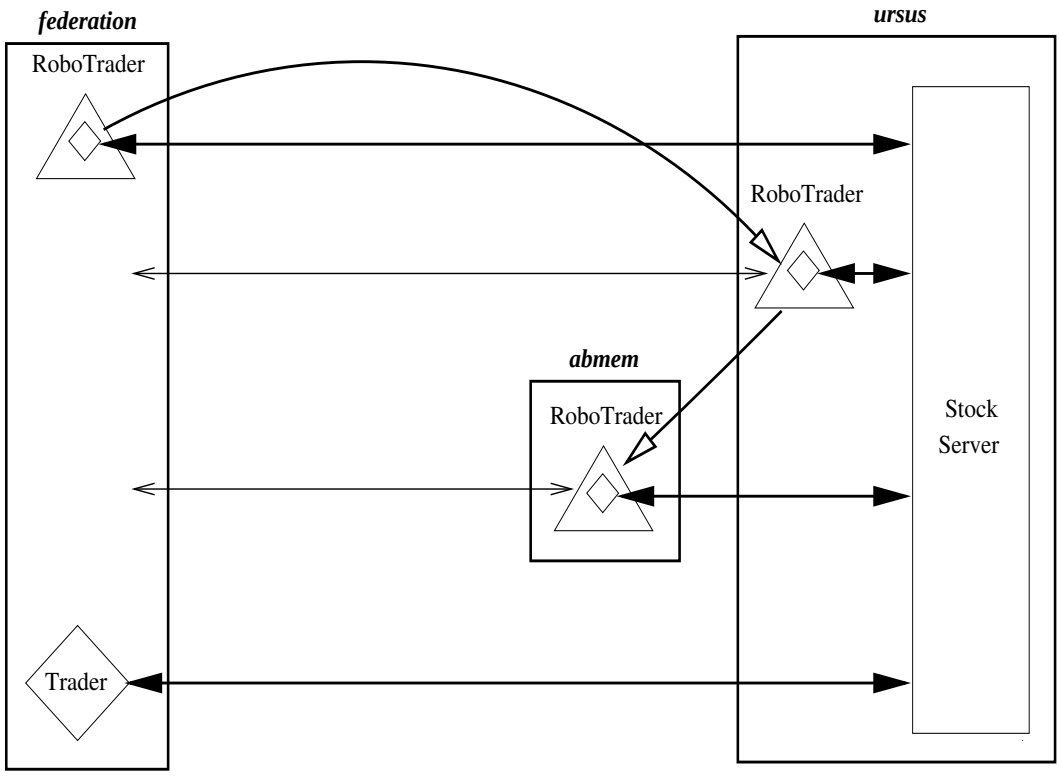

Figure 17: Experimental Setup

standing performance without requiring the server to accommodate agents, as long as there exists an agent server nearby.

Table 2 summarizes all of these results.

5.4 Conclusions

The first experiment shows that the agent paradigm achieves similar, although slightly inferior, performance when compared with traditional RPC: the price of flexibility. However, attempts to create flexible applications without agents result in significantly inferior and variable performance.

The RoboTrader, the outcome of the pattern-based approach to application design, managed to parley these performance benefits into a flexible, scalable agent application. The RoboTrader was shown as being able to cater to the client's needs while adapting to agent-ignorant servers by migrating

RoboTrader Host	DayTrader Success Rate	95% Confidence Interval
cmu	50.8%	± 1.4%
abmem	100.0%	± 0.0%
ursus	100.0%	± 0.0%

Table 2: Success Rates for the RoboTrader vs. the Trader

to a friendly agent server near the server. In the worst case scenario with no supporting agent servers, the RoboTrader achieved a success rate similar to the Trader itself. Thus, the RoboTrader's only costs were in its construction and deployment, and not during the actual execution.

As for said construction and deployment, the pattern-based approach seemed to facilitate this process. After developing one agent application(the Shepherd) and the traditional Trader application, the structural diagrams logically pointed towards combining these two to create a Monitor-based extension, the RoboTrader. The only changes needed to create the performance-enhancing RoboTrader were the arguments to the Shepherd program. The Shepherd was given the Trader (and its arguments) as its first argument, and a prioritized list of potential hosts(*ursus* first, *abmem* second, and *federation* third) as its second argument. Consequently, through the use of only two arguments, a scalable, flexible agent-based version of the Trader with significantly greater performance was constructed.

6 Conclusions

Our experiences confirm the idea that agent usage patterns provide an elegant means for developing scalable next-generation applications, promoting flexibility, performance, reliability, and security. We believe that a bottom-up pattern-based approach will ensure that these advantages are kept in focus, making it easier to create more effective network-based applications of the next generation.

To date, we have identified several fundamental agent patterns, and we have developed prototypes that have been used as bases for developing some small-scale applications. Our next steps include the further development of prototypes for other agent usage patterns, and to more systematically evaluate the hypothesis that through the use of these patterns, one can scale performance, reliability, and security. Part of this step will involve devising metrics to quantify the levels of scalability achieved. While quantifying levels of performance is fairly straightforward and was demonstrated in our experiments, levels of reliability and security are not so clearly delineated.

References

[1] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, D. Patterson. "The Interaction of Parallel and Sequential Workloads on a Network of Workstations", UC Berkeley Technical Report CS-94-838, Nov. 1994.

[2] S. Belmon, B. Yee. "Mobile Agents and Intellectual Property Protection," *Proc. Second Int'l Workshop on Mobile Agents 98*, 1998.

[3] M. Bever, "Distributed Systems, OSF DCE, and Beyond." *DCE - The OSF Distributed Computing Environment: Client/Server Model and Beyond* Springer-Verlag, 1993.

[4] A. Bieszczad, T. White, B. Pagurek, "Mobile Agents for Network Management." *IEEE Communications Surveys*, Sept. 1998.

[5] A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls." *ACM Trans. on Comp. Sys.*, Jan. 1984, pp. 39–59.

[6] S. Bryson. "Introduction [to Virtual Reality]", Implementing Virtual Reality, ACM SIGGRAPH '93 Course 43 Notes, 1993.

[7] S. Bryson, Y. M. Gerald. "The Distributed Windtunnel." Proc. Supercomputing '92, pp. 275–284.

[8] S. Cen, C. Pu, R. Staehli, C. Cowan, J. Walpole, "A Distributed Real-Time MPEG Video Audio Player." *Proc. of the 5th Int'l Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV '95)*, pp. 151–162.

[9] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, G. Tsudik. "Itinerant Agents for Mobile Computing." Technical Report RC 20010, IBM T. J. Watson Resesarch Center, Mar. 1995.

[10] W. Cockayne, M. Zyda, "Mobile Agents" Manning Publications Co, 1998.

[11] S. Franklin, A. Graesser. "Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents," Institute for Intelligent Systems, University of Memphis, 1996.

[12] E. Gamma, R. Helm, R. Johnson, R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[13] W. Gibson. "Neuromancer", Ace Books, 1984.

[14] R. Gray, D. Kotz, S. Nog, D. Rus, G. Cybenko, "Mobile agents for mobile computing," *Proc. of the 2nd Aizu Int'l Symp. Parallel Algorithms/Architectures Synthesis*, Fukushima, Japan, Mar. 1997.

[15] R. Gray. "Agent Tcl: A flexible and secure mobile-agent system." PhD Thesis, Dept. of Computer Science, Dartmouth College, Jun. 1997.

[16] R. Gray. "Agent Tcl: A flexible and secure mobile-agent system." *Proc. of the 4th Annual Tcl/Tk Workshop*, Jul. 1996.

[17] R. Gray, G. Cybenko, D. Kotz, D. Rus. "D'Agents: Security in a multiple-language, mobile-agent system." In *Mobile Agents Security*, Springer-Verlag, 1998.

[18] C. Harrison, D. Chess, A. Kershenbaum, "Mobile Agents: Are They a Good Idea?" IBM Research Report, Mar. 1995.

[19] D. Lange, M. Ishima. *Program and Deploying Java Mobile Agents with Aglets.*, Addison-Wesley, 1998.

[20] D. Johnasen, R. van Renesse, F. Schneider, "An introduction to the TACOMA distributed system", Technical Report 95-23, Dept. of Computer Science, University of Tromso, Jun. 1995.

[21] D. Johnasen, R. van Renesse, F. Schneider, "Operating system support for mobile agents." *Proc. of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[22] D. Johansen. "What TACOMA Taught Us.", talk. Paper available with F. Schneider and R. van Renesse as co-authors at http://www.tacoma.cs.uit.no/papers/taughtus.html.

[23] M. Litzkow, M. Livny, M. Mutka. "Condor – A Hunter of Idle Workstations," *Proc. of the 8th Int'l Conference of Distributed Computing Systems*, pp. 104–111, Jun. 1988.

[24] L. Mummert, M. Ebling, M. Satyanarayanan. "Exploiting Weak Connectivity for Mobile File Access." *Proc. of the 15th ACM Symposium on Operating Systems Principles*, Dec. 95.

[25] R. Pausch. "Virtual Reality on Five Dollars a Day," ACM SIGCHI: Human Factors in Computing Systems, Apr. 1991, pp. 265–270.

[26] H. Peine, T. Stolpmann. "The Architecture of the Ara Platform for Mobile Agents". *Proc. of the 1st Int'l Workshop on Mobile Agents*, Apr. 1997.

[27] M. Satyanarayanan, B. Noble, P. Kumar, M. Price. "Application-Aware Adaptation for Mobile Computing," *Operating Systems Review*, Jan. 95.

[28] F. B. Schneider. "Towards fault-tolerant and secure agentry". *Proc. of the 11th Int'l Workshop on Distributed Algorithms*, 1997

[29] J. Smith, et al., "SwitchWare: Accelerating Network Evolution." Technical Report MS-CIS-96-38, CIS Department, University of Pennsylvania, 1996. Also as http://www.cis.upenn.edu/ jms/white-paper.ps.

[30] N. Stephenson, "Snow Crash", Bantam Spectra, 1993.

[31] M. Straser, J. Baumann, F. Hohl. "Mole – a Java based mobile agent system", *2nd ECOOP Workshop on Mobile Object Systems*, pp. 28–35, Jul. 1996.

[32] J. Tardo, L. Valente. "Mobile agent security and Telescript", *41st International Conference of the IEEE Computer Society (CompCon '96)*, Feb. 1996.

[33] D. Tennenhouse, D. Wetherall. "Towards an Active Network Architecture", *Computer Communication Review*, 26(2), Apr. 1996.

[34] T. Vallillee, "A Guide to SNMP and CMIP." Available from: http://www.geocities.com/ SiliconValley/Horizon/4519/snmp.html.

[35] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments". *IEEE Communications Magazine* 14(2), Feb. 1997.

[36] D. Wetherall, D. Tennenhouse. "The ACTIVE IP Option", *7th ACM SIGOPS European Workshop*, 1996.

[37] J. E. White. "Telescript Technology: The Foundation for the Electronic Marketplace". General Magic White Paper, General Magic, Inc., 1994.

[38] B. Yee. "A Sanctuary for Mobile Agents", UCSD Technical Report CS97-537, Apr. 1997.