A Reliable Connection Migration Mechanism for Synchronous Transient Communication in Mobile Codes

Xiliang Zhong and Cheng-Zhong Xu
Department of Electrical & Computer Engg.
Wayne State University, Detroit, Michigan 48202
{xlzhong, czxu}@wayne.edu

Abstract

With the increasing popularity of network applications, mobile codes are playing a key role in providing scalable services. Due to their mobility nature, it is a challenge to support a synchronous transient communication between mobile objects. This paper presents a reliable connection migration mechanism that allows mobile objects in communication to remain connected during their migration. This mechanism supports concurrent migration of both endpoints of a connection and guarantees exactly-once delivery for all transmitted data. In addition, a mobile code access control model is integrated to ensure secure connection migration. This paper presents the design of the mechanism and a reference implementation, namely NapletSocket, over Java Socket in a mobile agent system. Experimental results show that the NapletSocket incurs a moderate cost in connection setup, mainly due to security checking, and marginal overheads for communication over established connections. Furthermore, we investigate the impact of agent mobility on communication performance via simulation. Simulation results show that NapletSocket is effective and efficient for a wide range of migration and communication patterns.

Keywords: Mobile Codes, Mobile Agent, Synchronous Communication, Persistent Connection, Connection Migration

1 Introduction

Mobile code, as its name implies, refers to programs that function as they are transferred from one machine to the other. The concept of code mobility is widely used in today's network services for load balancing, fault resilience, system administration, improving data access locality, etc. A special form of migration is mobile agent that has as its defining trait the ability to travel autonomously, carrying its code, as well as data and running state. Because of the unique property, mobile agents have been the focus of much speculation and hype in the past decade; see [10] for a recent comprehensive review.

In mobile agents based computing, it is often necessary for agents to communicate with each other to work efficiently. Conventional technologies for remote inter-agent communication are through a mailbox-like *asynchronous persistent* communication mechanism due to the requirement for agent au-

tonomy [2]. That is, an agent can send messages to others no matter its communication parties exist or not. The messages in transmission are often forwarded in support of agent migration.

Asynchronous persistent communication plays a key role in many distributed applications and is widely supported by existing mobile agent systems; see [19] for a review of location independent communication protocols between mobile agents. A common asynchronous communication mechanism works by two steps. First a message is sent to an intermediate, such as a proxy or a mail-box. Then the message is forwarded to the receiver agent. In this communication model, it is hard for the sender agent to determine whether and when the receiver gets the message. Thus it is not always appropriate and sufficient for applications that require agents to closely cooperate. For example, in the use of mobile agents for parallel computing [22], cooperative agents need to be synchronized frequently during their lifetime. A *synchronous transient* communication mechanism would keep the agents working more closely and efficiently. TCP Socket is a solution for instantaneous communication in distributed applications. However, the traditional TCP protocol has no support for mobility because it has been designed with the assumption that the communication peers are stationary. To guarantee message delivery in case of agent migration, a connection migration scheme is desirable so that an established socket connection would migrate with the agent continuously and transparently.

There are recent studies on mobile TCP/IP in both network and transport layers to support the mobility of physical devices in the arena of mobile computing. We refer to this type of mobility as physical mobility, in contrast to logical mobility of codes. Representatives of the protocols include Mobile IP [11] in network layer, MSOCKS [12],TCP-R [9], M-TCP [18] and Migrate [17] in transport layer. Although these protocols provide feasible ways to link moving devices to network, they have no control over the logical mobility. Mobile agents may also run on wired networks that have no support for mobile TCP/IP.

Mobile agent systems are usually organized as a middleware. Agent connection migration requires support of session-layer implementations in the middleware. In the past, a few session-layer connection migration mechanisms were proposed. Examples include Persistent Connection [24], Mobile TCP [16], and MobileSocket [15]. However, none of them was targeted at agent mobility. Agent related connection migration involves two unique reliability and security problems. Since both the end points of a connection would move around, a reliable connection migration needs to do more for exactly-once delivery for all transmitted data. Security is a major concern in agent-oriented programming. Socket is one of the critical resources and its access must be fully controlled by agent servers. Agent-oriented access control must be enforced during the setup of connections. Connection migration is vulnerable to eavesdropper attacks. Additional security measures are needed to protect transactions over an established connection from any malicious attacks due to migration.

In this paper, we present the design and implementation of an integrated mechanism that deals with reliability and security in connection migration. It provides an agent-oriented socket programming interface for location-independent socket communication. The interface is implemented by a socket controller that guarantees exactly-once delivery of data, including data in transmission when the communicating agents are moving. To assure secure connection migration, each connection migration is associated with a secret session key created during connection setup. We prototyped the mechanism as a

NapletSocket component in Naplet mobile agent system. Naplet [20] is a featured mobile agent system we developed in house for educational purposes. It supports a mailbox-based PostOffice mechanism with asynchronous persistent communication. NapletSocket provides a complementary mechanism for synchronous transient communication. Furthermore, we conducted simulations to study the impact of agent mobility on communication performance. Simulation results show that NapletSocket is effective and efficient for a wide range of agent migration and communication patterns.

The remainder of the paper is organized as follows. Section 2 gives an overview of the design of NapletSocket. Reliability, security concerns and other issues are discussed in Section 3. Section 4 presents experimental results of NapletSocket. Section 5 presents a communication performance model of NapletSocket and simulation results on the impact of agent mobility. Related works are summarized in Section 6. Section 7 concludes the paper with remarks on future work.

2 NapletSocket: A Connection Migration Mechanism

2.1 NapletSocket Architecture

The NapletSocket connection migration mechanism provides an interface similar to Java Socket. It comprises of two classes *NapletSocket(agent-id)* and *NapletServerSocket(agent-id)*. They resemble Java Socket and ServerSocket in semantics, except that the NapletSocket connection is agent oriented. It is known that Java Socket/ServerSocket establish a connection between a pair of endpoints in the form of (Host IP, Port). Due to security reasons, an agent is not allowed to specify a port number for its pending connection. Instead, it is the underlying NapletSocket system that allocates ports to the connection based on resource availability and access permissions. Naplet system contains an agent location service that maps an agent ID to its physical location. This ensures location transparent communication between agents. Once the connection is established, all communication is through the connection and no more location service is needed.

To support connection migration, the NapletSocket system provides two new methods *suspend()* and *resume()*. They can be called either by agents for explicit control over connection migration, or by Naplet docking system for transparent migration.

Figure 1 shows the NapletSocket architecture. It comprises of three main components: data socket, controller and redirector. The component of data socket is the actual channel for data transfer. It is associated with a pair of send/receiver buffers to keep undelivered data. The controller is used for management of connections and operations that need access right to socket resources. The redirector is used to redirect socket connection from a remote agent to a local resident agent. Both controller and redirector can be shared by all NapletSockets so that only one pair is necessary.

During connection establishment, the controller of the client agent sends a request to the counterpart at the server. After the request is acknowledged, the client connects to the redirector at the server side and the connection is then handed to the desired agent. After a connection is established, the two agents communicate with each other through accessing the data socket, no matter where their communication parties are located. Under the hood are a sequence of operations by the NapletSocket library.

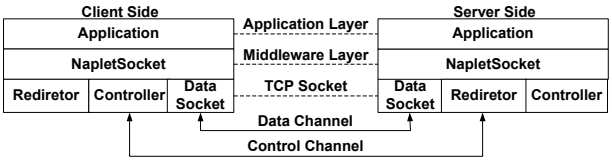


Figure 1: NapletSocket Architecture.

State	Description
CLOSED	Not connected
LISTEN	Ready to accept connections
CONNECT_SENT	Sent a CONNECT request
CONNECT_ACKED	Confirmed a CONNECT request
ESTABLISHED	Normal state for data transfer
SUS_SENT	Sent a SUSPEND request
SUS_ACKED	Confirmed a SUSPEND request
SUSPEND_WAIT	Wait in a suspend operation
SUSPENDED	The connection is suspended
RES_SENT	Sent a RESUME request
RES_ACKED	Confirmed a RESUME request
RESUME_WAIT	Wait in a resume operation
CLOSE_SENT	Sent a CLOSE request
CLOSE_ACKED	Confirmed a CLOSE request

Figure 2: States in NapletSocket transitions.

The underlying data socket is first closed, when the NapletSocket takes a suspend action before agent migration. During agent migration, no data can be exchanged since the connection is suspended. After the agent lands on the destination, the NapletSocket system will resume the connection by connecting to the redirector at the other side. The data sockets of both client and server are then updated and new input/output streams are re-created atop of the socket.

Since there is no need for message forwarding, communication over NapletSocket is efficient. In Section 4, we will show suspend and resume operations incur marginal overheads to keep connections persistent.

2.2 State Transitions

The design of NapletSocket can be described as a finite state machine, extended from the TCP protocol. It contains 14 states, as listed in Figure 2. The states in bold are newly added to the standard TCP state transitions. The states SUSPEND_WAIT and RESUME_WAIT are used only for concurrent connection migrations, which will be covered in later sections. A NapletSocket connection is in one of these states. Certain action will be taken when an appropriate event occurs, according to the current state of the connection. There are two types of events: calls from local agents and messages from remote agents. Actions include sending messages to remote agents and calling local functions.

Figure 3 shows the state transitions of a NapletSocket connection. The solid lines show the transitions of clients connecting to servers and the dotted lines are for the servers. Details of the open,

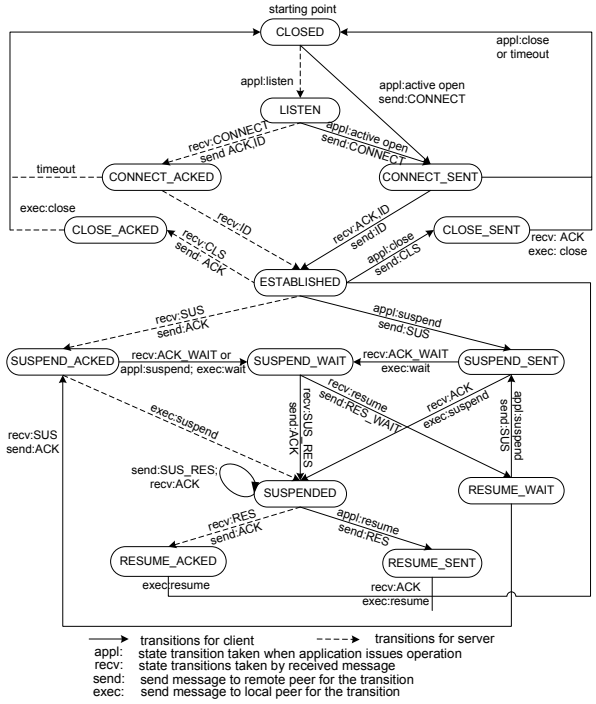


Figure 3: NapletSocket state transitions diagram.

suspend, resume and close transactions are as follows.

Open a connection. Both client and server are initially at the CLOSED state. When an agent does an active open, a CONNECT request is sent to the server and the state of the connection changes to CONNECT_SENT. If the request is accepted, the client side NapletSocket receives an ACK and a socket ID to identify the connection. Then it sends back its own ID and the state changes to ESTABLISHED.

Connection in server side switches to the LISTEN state once an agent does a listen. When a CONNECT request comes from a client, the server acknowledges it by sending back an ACK and a socket ID. It then changes to the CONNECT_ACKED state. After the socket ID of the client side is received, it switches to ESTABLISHED and the NapletSocket connection is established. Now data can be transferred between the two peers as normal socket connection.

Suspend/Resume a connection. After a connection is established, either of the two parts may suspend it. The one who wants to suspend a connection invokes the suspend interface and a SUS is sent to the peer. If the request is acknowledged, an ACK is sent back and triggers the action of closing underlying input/output streams and data socket. The connection state then switches to SUSPENDED.

When the other side of NapletSocket receives the SUS message, it sends back an ACK if it agrees to suspend. Then it closes the underlying connection. After that, the state for this peer also changes to SUSPENDED. Now connections at both peers are suspended. No data can be exchanged in this state.

At the SUSPENDED state, when either of the agents decides to resume the connection, it invokes the resume interface. The resume process first sets up a new connection to the remote redirector and sends

a RES message. If an ACK is received, it then resumes the connection and the state switches back to ESTABLISHED. Once the remote peer in the SUSPENDED state receives a resume request, it first sends back an ACK. Then the redirection server hands its connection to the desired NapletSocket and new input/output streams are created. After that, both peers change back to the ESTABLISHED state.

Close a connection. In either the ESTABLISHED or the SUSPENDED state, if an agent decides to close the current connection, it invokes the close interface and the NapletSocket does an active close by sending a CLS(CLOSE) request to the peer. After the request is acknowledged, local data socket is closed. The other side of the connection closes passively after receiving a CLS request. It first acknowledges the request and then closes the underlying socket and streams. At the time, data socket at both sides is closed and the state changes to CLOSED.

3 Design Issues

3.1 Transparency and Reliability

If there is any connections setup before agent migration, the connections should be migrated transparently. We first discuss the case for only one connection. Then we extend it to multiple connections. The main approach for connection migration is to use a data socket under NapletSocket. Each time the agent migrates, the underlying data socket is closed before migration and updated to a new data socket after migration. When two agents migrate freely, it is possible that migration happens at the same time when there are data being transferred and in this case, the data may fail to reach their destinations. It is the presence of mobility causes a problem for reliable communication. Furthermore, two agents may migrate simultaneously which makes it more difficult to achieve reliability.

To guarantee that messages can be finally delivered, we added an input buffer to each input stream and wrapped them together as a NapletInputStream. To suspend a connection, the operation retrieves all currently undelivered data into the buffer before it closes the socket. The data in the NapletInputStream migrate with the agent. When migration finishes and the connection is resumed at the remote server, a read operation first reads data from the input buffer of its NapletInputStream. It doesn't read data from socket stream until all data from the buffer have been retrieved.

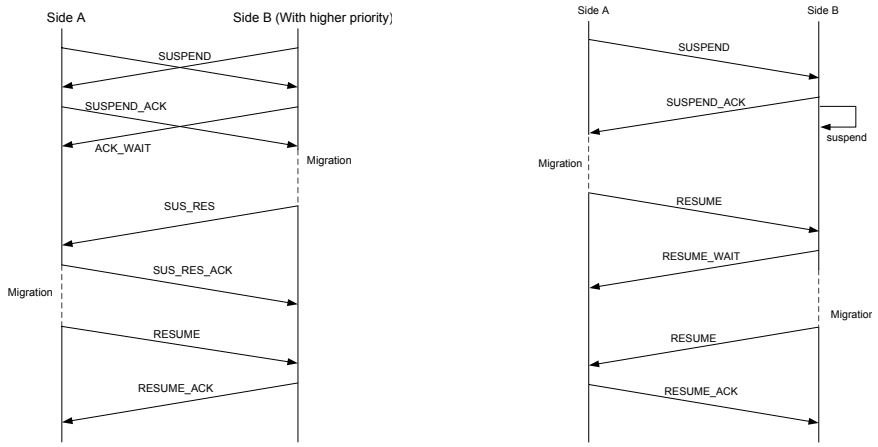
In the case both agents of a connection want to move simultaneously, there is a problem since the resume operation from one agent only remembers the previous location of the other agent. It is not difficult for NapletSocket to know where the next host is to support two migrations at the same time. The problem is an agent may migrate frequently in the network. A resume operation may have to chase a mobile agent if two agents are allowed to migrate concurrently. In order to avoid the chasing problem and provide a simplified solution, we delay the migration of one agent if two are issued at the same time, which means only one can migrate at a time and the other can not leave until the first one finishes. Therefore agents migrations occur sequentially although they are issued simultaneously. But from the viewpoint of high level applications, the underlying sequential migration is transparent and there is no restriction for agents migration.

To delay one operation when two suspend operations are issued around the same time, two states SUSPEND_WAIT and RESUME_WAIT are introduced. One of them is delayed and the state of the connection is put into the SUSPEND_WAIT state. At this state, the operation is blocked until the agent finishes migration and sends a notification message. Then the connection is switched to the SUSPENDED state and the second agent can migrate to another host.

RESUME_WAIT is the state when a resume operation is blocked. In the case of concurrent migration, when one of the peer finishes migrating, it invokes the resume operation to update underlying connection and I/O streams. If we approve the resume operation finish as usual, then a connection is to be established between the agent who has just finished migration and the one who is to migrate. The connection switches to ESTABLISHED state. But the connection has to be suspended again due to the second agent migration and the switches of states from SUSPENDED to ESTABLISHED and back is not necessary. The purpose of RESUME_WAIT is to prevent the state transition from SUSPEND to ESTABLISHED. During the resume operation, instead of establishing a new socket connection, we change the state of the connection to RESUME_WAIT and block the resume operation. The resume operation will be signaled after the other agent finishes migration. Since no new connection is established after the first agent migration, there is no need to suspend the connection before the second agent migration. By using this RESUME_WAIT state, we save time for a suspend operation and part of a resume operation.

During concurrent agent migration, it is possible for socket controller to issue a suspend operation and at the same time receive a SUSPEND request from the other side of the connection. Depending on when a suspend operation is issued, we classify the problem into two types. One is overlapped concurrent connection migration when the suspend operation is issued before an acknowledgement for the SUSPEND request from the other side is sent; the other is non-overlapped when the operation is issued after an acknowledgement for SUSPEND has been sent and response for the SUSPEND is still in progress.

Overlapped concurrent connection migration. In the first case, both sides of a connection issue SUSPEND requests at about the same time and neither receives an acknowledgement before the SUSPEND request from the other side arrives. Each side has to decide whether to grant the request. To approve one and only one request, we give priority to one of the agents to let it migrate first. The other one is delayed until the first finishes. A time sequence for this case is in Figure 4(a). At the beginning of the sequence, there is a connection between side A and B. Both of them issue a SUSPEND request at about the same time. We assume side B has a high priority. Side B receives the request and since it has sent a SUSPEND request, this is treated as concurrent connection migration. Side B sends back an ACK_WAIT to delay the migration. Same situation is for side A. But side A always acknowledges a SUSPEND request since it has a low priority. Then agent in side B migrates and the state of the connection in side A is switched to SUSPEND_WAIT. After finishing migration, side B informs side A with a SUS_RES to continue the blocked suspend operation. Side A confirms with an acknowledgement and finishes the issued suspend operation. Then the connection is kept in SUSPENDED state for both side A and B. At the time, agent in side A may migrate to another host. After that, it resumes the connection by normal resume operation. Then the connection in both sides returns to the ESTABLISHED state.



(a) Overlapped migration.

(b) Non-overlapped migration.

Figure 4: Time sequence of concurrent connection migration.

Non-overlapped concurrent connection migration. In the second case, one side of a connection issues a suspend operation after it acknowledges a SUSPEND request from the other side and the previous request hasn't finished. The second suspend operation shouldn't continue until the first one finishes. This is another case for concurrent connection migration and we call it non-overlapped migration since the processing of the two suspend operations are not overlapped. In this case, it doesn't matter which side of the connection has a high priority since the acknowledged request has to finish first. A time sequence is presented in Figure 4(b) for this case. At first, side A issues a SUSPEND request and side B replies with an acknowledgement. Then side A suspends the connection and starts to migrate. Before it finishes, agent in side B decides to migrate so a suspend operation is invoked from socket controller in side B. Since this request couldn't get approved, so instead of sending a SUSPEND request to side A, we switch the connection state in side B to SUSPEND_WAIT. After agent in side A finishes migration, it sends a RESUME request to resume the connection. Because side B has a blocked suspend operation to finish, it replies with a RESUME_WAIT message to block the resume operation and continue with its suspend operation. After that, connection in side B is in the SUSPENDED state and in side A it is kept in the RESUME_WAIT state. Then agent in side B can migrate and a normal resume operation is used to switch the state to ESTABLISHED after the migration.

Priority. We give priorities to one of the agents when two want to migrate at the same time. A question is how to determine which one is assigned a high priority. A simple solution is to determine the priorities according to their roles in the connection. For example, we can give a high priority to all connections established from ServerSocket. But this solution is prone to deadlock in an environment with multiple agents connected with each other. For example, in a configuration of three agents X, Y, and Z which are clients of Y, Z, and X, respectively. This forms a circular waiting list if all of the agents want to move at the same time. To prevent deadlock in simultaneous migration, we determine the migration priority of each agent based on its unique agent ID. During connection setup, a hash

function is applied to each agent ID and generate hash values for both agents. We assign their priorities according to their ordered hash values.

3.2 Multiple Connections

In preceding discussions, we focused on concurrent agents migration for one connection. In the case multiple connections are established before agents migration, all connections should be suspended. When suspending all these connections, it is possible for them to be suspended in different orders. For example, suppose there are two connections for agents in side A and B, represented as #1 and #2. Side A may suspend connections in the order of #2 #1 and side B in the order of #1 #2. When the two sides issue suspend operations simultaneously, it is possible for side A to suspend connection #2 while at the same time side B is suspending #1. Either these concurrent suspend operations needs to be delayed because they are operating on different connections. Thus both of the connections are successfully suspended. When suspending the second connection, side A on #1 while side B on #2, both sides will find the connection has already been suspended. By default a suspend operation needs to do nothing for a suspended connection. Therefore both sides successfully suspend their connections and agents migrations happen at the same time. As a result, either of the connection knows where the other peer is after landing on a new host.

To have only one agent migrate at a time, suspend operations for one of the connections must be delayed. We achieve this by giving different responses to a suspend operation when it is operating on a connection that is in the SUSPENDED state. Suspend operations are distinguished by whether they are issued locally or invoked by remote messages. When controller issues a suspend operation due to agent migration, it is local suspending. When the connection is suspended on receiving a SUSPEND request, it is remote suspending. During a suspend operation, if the connection has already been suspended remotely which means agent migration is happening in the other side, we decide whether to continue or block depending on priority of the peer. If it has a low priority, we block the suspend operation; if it has high priority, we finish the operation without further actions since it is already in the SUSPENDED state. The blocked suspend operation will get notified when the one with high priority finishes migration.

Figure 5 gives an illustration of the protocol. In this example, we assume side B has a high priority. Initially, side A issues a SUSPEND request for connection #2 and side B issues a SUSPEND request for connection #1 at about the same time. Because the operations are on different connections, they both get approved. Then connection #2 in side A and connection #1 in side B become locally suspended. Connection #1 in side A and connection #2 in side B become remotely suspended. After that, side A issues a suspend operation for connection #1. It is blocked because the connection is suspended remotely and side A is in a low priority. The same thing happens with connection #2 for side B. But because side B has a high priority, the suspend operation returns without further actions. Then agent in side B proceeds with migration while connection in side A is in the SUSPEND_WAIT state. After agent in side B lands on a new host, side B issues a resume operation for the blocked suspend in side A for connection #1 and resume connection #2. Side A sends back RESUME_WAIT for the resume connections because it is to migrate. After receiving the message, connection in side B switches to the

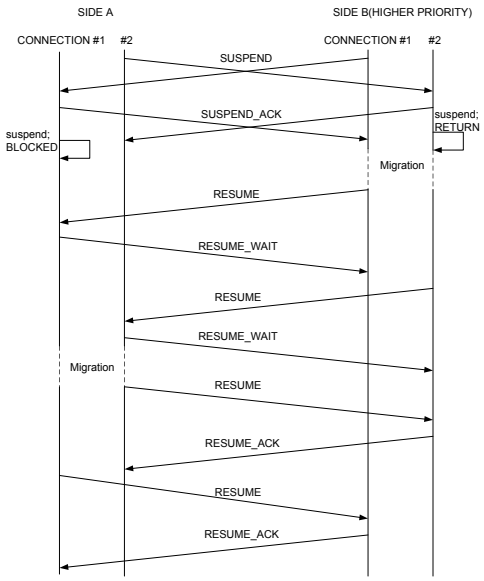


Figure 5: Time sequence of concurrent agent migration with multiple connections.

state of RESUME_WAIT. After agent migration finishes in side A, RESUME messages are sent from side A to side B for both connection #1 and # 2 and the two connections switch to the ESTABLISHED state.

3.3 Security

Security is always a basic and direct concern in mobile agent systems. NapletSocket addresses security issues in two aspects. First, the agent should not be able to cause any security problems to the host it resides, either at the original host or at the host it migrates to. Second, the connection itself should be secure from possible attacks like eavesdropping. More specifically, a connection can only be suspended/resumed by the one who initially creates it.

Regarding the first issue, any explicit requests to create a Socket or ServerSocket from an agent are denied. Permissions are only granted to requests from the NapletSocket system. Thus the only way for an agent to use socket resources is through the service provided by the mobile agent system. Now the problem becomes whether we can deny permission if a request is from an agent and grant it if it is from the system.

This problem can be solved by user-based access control introduced in the latest JDK security mechanism. It allows permissions to be granted according to who is executing the piece of code (subject), rather than where the code comes from (codebase). A subject represents the source of a request such as a mobile agent or NapletSocket controller. By denying access requests from the subject of agents for socket resources and granting them to local users such as administrators, we achieve our security goal in a simple and clear manner. In mobile agent applications, an agent subject has no permissions to access local socket resources by default. When it needs access to a socket resource, it submits a request to a proxy service in NapletSocket controller. The proxy authenticates the agent and checks access permissions. After the security check passes, a NapletSocket or NapletServerSocket will

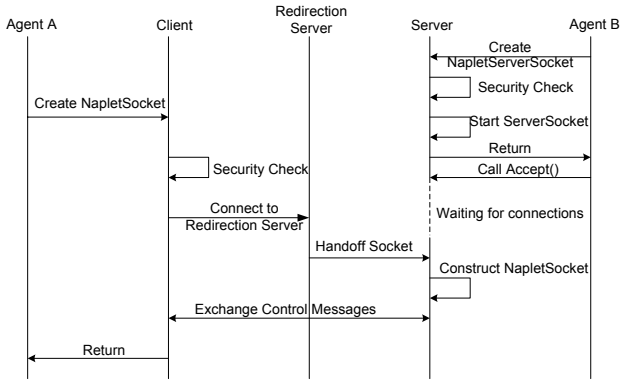


Figure 6: Socket handoff in connection setup.

be created by the proxy and returned to the agent. More details about agent-oriented access control in Naplet system can be found in [21].

Regarding the second issue, connection migration can be realized by the use of a socket ID. However, a plain socket ID couldn't prevent a third party from intercepting the information and exercising eavesdropping attacks. To this end, we applied Diffie-Hellman key exchange protocol [5] to establish a secret session key between the pair of communicating agents at the setup stage of a connection. Any subsequent requests for suspend, resume, and close operations on the connection must be accompanied with the secret key. Such requests will be denied unless their keys are verified by remote peers. Since the key generated by Diffie-Hellman protocol is hard to break, NapletSocket connections are protected from eavesdropping attacks.

3.4 Socket Handoff

As we know in NapletSocket, when a client connects to a server, it uses an agent ID to specify the destination. We need to find both host name and port number from the ID. To find host name, we can keep records of traces of agents and locate the host according to its ID. To find port number, we need to send a query message to the server. The server has to maintain a table indicating which agent uses which port and return the port to client. Then the client can start a connection. In fact, these operations can be saved using the socket handoff technique. By connecting to the redirector at the server and indicating the desired agent. The server looks up which NapletServerSocket this request is for and redirects the current socket to it. Then the NapletServerSocket gets notified from blocking and constructs a NapletSocket according to the data socket it receives. This mechanism can save a round trip time in querying host name and port number and there is no need for the server to maintain a table mapping ports to agents. Figure 6 shows the sequence diagram for socket handoff in connection setup.

Similar mechanism also applies when resuming a connection. In this case, the client connects to redirection server at the other side and sends a request to resume. The server then hands the socket connection to the suspended NapletSocket and wakes it up. Finally the notified NapletSocket updates its underlying data socket.

3.5 Control Channel

It is necessary to exchange control messages during state transitions of a NapletSocket connection. From a performance perspective, we used a separate channel for control messages and chose UDP as the transport layer protocol. Regarding the omission failures and ordering problems caused by UDP, we adopted a retransmission mechanism to provide reliable delivery on top of UDP. The basic idea is to use retransmission in case of failure. After sending a control message, the sender starts a retransmission timer and waits for an ACK from the receiver. If an ACK is received before timeout, the timer is cancelled. If not, the message is retransmitted and a new timer for the message is set. Sequenced numbers are used to relate a reply to the corresponding request.

4 Experimental Results of NapletSocket

In this section, we present an implementation of NapletSocket and its performance in comparison with Java Socket. All experiments were conducted in a group of Sun Blade 1000 workstations connected by a fast Ethernet. We first show the effectiveness and overhead of the implementation, focusing on the cost of its underlying operations such as open,suspend and resume. We then evaluate the overall communication performance under various migration and communication patterns.

4.1 Effectiveness of Reliable Communication

The first experiment gives a demonstration of reliable communication using NapletSocket. A stationary agent A keeps sending messages at a rate of one millisecond to a mobile agent B. Each message contains a counter, indicating the message order. On the receipt of a message, agent B echoes the message counter back to A. Reliable communication requires the mobile agent B receives the messages in the same order as they are sent.

Figure 7 shows the traces of the message counters received by the mobile agent B over the time. Agent B migrates at 10th, 20th, 30th milliseconds. The dark dots show the messages read from the socket stream and the light dots are messages into or from message buffer in NapletSocket. In the first migration point, agent B migrates before it receives all of the messages in transmission. The undelivered three messages (7, 8, 9) are kept in the message buffer of NapletSocket. They are transferred together with agent B under the support of NapletSocket at the sender side and delivered to agent B by the NapletSocket at the receiver side after B lands. Similarly, the third agent migration involves transferring of one message at the same time.

4.2 Cost of Primitive NapletSocket Operations

The second experiment focused on the cost of primitive operations defined in NapletSocket, including connection open, close, suspend, and resume. The cost of an operation refers to the interval between the time when it is issued and the time when it is completed.

Table 1: Latency to open/close a connection.

Connection Type	Open (ms)	Close (ms)
Java Socket	3.7	0.6
NapletSocket w/o security	18.2	12.5
NapletSocket with security	134.4	12.6

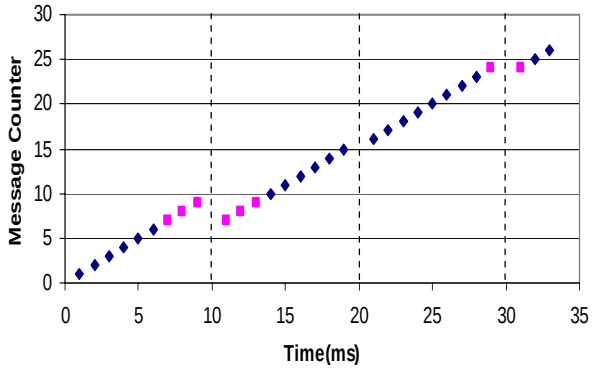


Figure 7: A Message trace demonstrating the effectiveness of reliable communication.

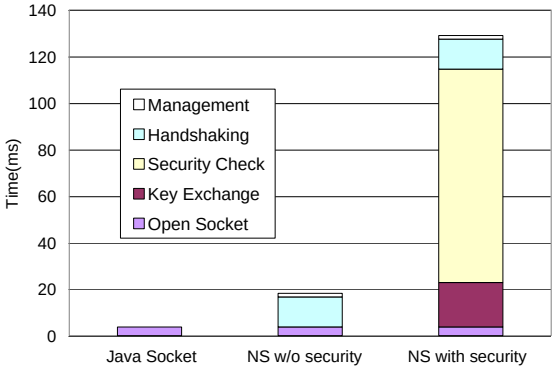


Figure 8: Breakdowns of the latency to open a connection.

We performed open and close operations with and without security checking for 100 times each. Table 1 shows the average time for each operation in different cases. For comparison, the costs for open and close operations in Java Socket are included. From the table, we can see that opening a secure mobile connection costs almost 40 times as much as that of a Java socket. Recall that establishment of a secure NapletSocket channel involves a number of steps: authentication, authorization, secret key exchange, handshaking and socket establishment. Figure 8 shows the breakdown of the cost. The figure shows that more than 80% of the time was spent on key establishment, authentication and authorization. The cost would reduce to 18.2ms without security support. It remains very high in comparison with the cost in Java Socket, but it is acceptable. It is because open a connection is a one-time operation and the connection remains alive once established, which means that cost of opening a connection is amortized over agent migration.

Similarly, we recorded the cost of 27.8ms and 16.9ms for suspend and resume operations, respectively. The cost is mainly due to the exchange of control messages (handshaking), which makes up about 50% for suspending and 70% for resuming. Suspending a connection also requires to check if there are any undelivered data in the input stream. Resuming a connection needs to set up a data socket and create I/O streams.

The benefit of provisioning a reliable connection can be seen by comparing the time required for re-opening a connection with that of suspend/resume. If we close a NapletSocket before migration and reopen a new one after that, the total cost involved is about 147ms. However, if we use suspend and resume instead, the cost is less than one third of the time for close and reopen operations. The total time saved by using suspending and resuming increases with the move of the agent.

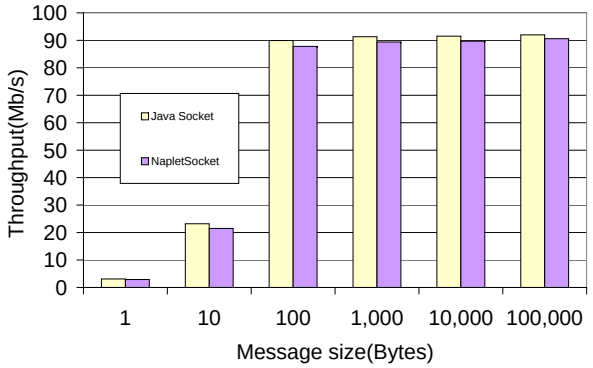


Figure 9: Throughput of NapletSocket vs. Java Socket.

4.3 NapletSocket Throughput

In the third experiment, we tested the NapletSocket throughput by the use of TTCP [4] measurement tool, in which a pair of TTCP test programs call Java Socket methods to communicate messages of different sizes as fast as possible. Because NapletSocket bears much resemblance to Java Socket in their APIs, we developed a simple adaptor to convert TTCP programs into NapletSocket compliant codes.

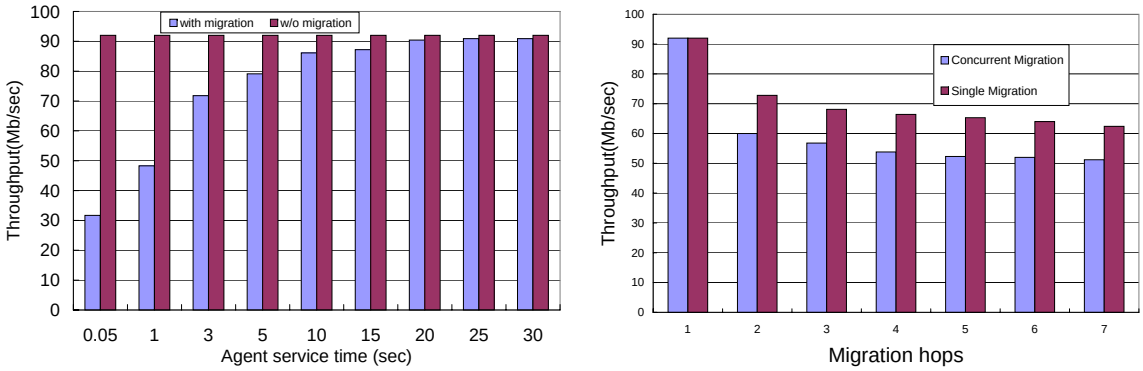
Figure 9 shows the NapletSocket throughput between two stationary agents. The throughput of Java Socket is included for comparison. Each data is an average of 7 independent runs. From the figure, we can see the NapletSocket throughput degrades slightly (less than 5%). This degradation is mainly due to synchronized access to I/O streams. With the increase of message size, the performance gap becomes almost negligible.

To test the impact of agent mobility on NapletSocket throughput, we designed two migration patterns, in which a pair of agents keep communicating to each other, while they are traveling around. One is *single migration* where one agent remains stationary and the other keeps moving at a certain rate. The other is *concurrent migration* in which both agents travel simultaneously along their own paths and communicate to each other at each hop.

Expectedly, the NapletSocket throughput tested by TTCP are dependent on agent migration frequency (i.e. service time at each hop). We refer to the total traffic communicated over a period of communication and migration time as *effective throughput*. Figure 10(a) shows the effective throughput with different migration frequencies in single migration pattern. In this experiment, we assumed a constant message size of 2K bytes.

From the figure, we can see that the measured effective throughput between stationary agents goes up to 92Mb/s. The throughput increases as the agents spent more time in communicating in each host, starting from 32Mb/s when the service time is one second to the maximum value when the agent stays in a host for more than 10 seconds. If an agent stays in a host for long enough time, the effective throughput gets very close to the one without migration. That implies that the effect of agent and connection migrations on throughput becomes negligible when an agent migrates at a low frequency.

Furthermore, we examined the impact of migration hops on throughput. Figure 10(b) shows the



(a) Impact of migration frequency.

(b) Impact of migration hops.

Figure 10: Effective throughput of NapletSocket under different agent migration patterns.

effective throughput as agents migrate in both single and concurrent migration patterns. In this experiment, service time was fixed to as large as 20 seconds per host so as to isolate the performance from the impact of migration frequency. From the figure, we can see that as an agent visits more hosts, the throughput drops, but at a very slow rate. This is expected because with the migration of the agent, more migration overheads incurred in the calculation of the effective throughput. Since the total time spent in data transferring also increases at a higher rate, the throughput decreases at a slower rate. For example, in the single migration pattern, the effective throughput drops by 6.2% as the mobile agent migrates to the 3rd host and by 2.5% when it travels to the 7th host.

From Figure 10(b), we can also see that the effective throughput in concurrent migration is smaller than that of single migration. It is because concurrent migration incurs more overheads. In the next section, we will have more discussions about the impact of migration concurrency on the performance.

5 Performance Model of Agent Mobility

The experiments in the preceding section assumed agents communicate to each other "as fast as possible". Also in the concurrent migration pattern, both agents actually migrated at the same fixed rate due to the assumption of constant service time at each host. The objective of this section is to investigate the impact of the communication rate as well as the agent migration concurrency on effective throughput of NapletSocket.

5.1 Performance Model

Consider two mobile agents, say A and B, which are connected via a NapletSocket connection. Both of them are traveling around a network at various rates. Without loss of generality, we assume agent B has a higher priority than A. At each host, the agents process their tasks for certain time and communicate with each other for synchronization, as shown in Figure 11. Associated with each agent migration is a connection migration.

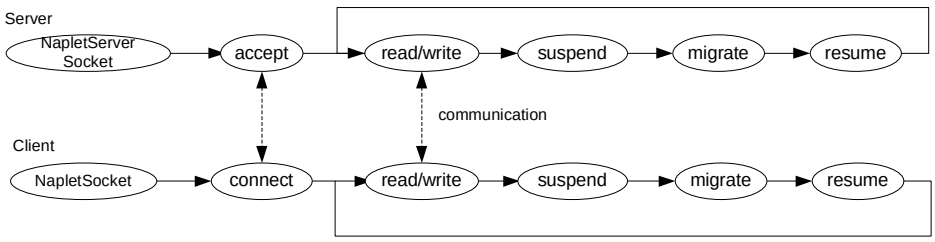


Figure 11: Migration/Communication Pattern using NapletSocket.

It is known the effective throughput of NapletSocket is determined by the cost for connection migration as well as the overhead for agent migration. Since the cost for agent migration is application-dependent (being dependent upon the code and state sizes), we develop a model for connection migration, denoted by $T_{c-migrate}$, instead.

Recall that a connection migration starts with a suspend request and ends with a resume operation. A suspend operation changes the state of a connection from ESTABLISHED to SUSPEND. A resume operation changes the states of a connection from SUSPEND back to ESTABLISHED. Denote $T_{suspend}$ and T_{resume} as the costs for suspend and resume operations, respectively. It follows that

$$T_{c-migrate} = T_{suspend} + T_{resume}. \tag{1}$$

Notice that the costs for suspend and resume operations are related to agent migration concurrency. In the case that only one endpoint of a connection is mobile, both $T_{suspend}$ and T_{resume} are constant. In the case of concurrent migration where both endpoints of a connection are mobile, $T_{suspend}$ and is dependent upon the ordering of their requests. Let t^a_{begin} and t^b_{begin} denote the request time of the two agents A and B and their interval $\tau = |t^a_{begin} - t^b_{begin}|$. If the interval τ is large enough for the first suspend to complete before the second suspend is issued, it becomes the case of single agent migration. Otherwise, the two suspend operations are performed concurrently.

As we discussed in Section 3.1, concurrent migration can be further distinguished between two cases: overlapped and non-overlapped. In the following, we analyze the cost for suspend operation in these two cases.

Overlapped concurrent migration. For agent B with a higher priority, its suspend cost $T^b_{suspend}$ is the same as in the single migration pattern. For agent A with a lower priority, its suspend operation couldn't finish until it receives a SUS_RES message from B, indicating the completion of B's migration, as shown in Figure 4(a). From the figure, we can see that the arrival time of SUS_RES at agent A

$$t_{sus\_res} = t^b_{begin} + T^b_{suspend} + T^b_{a-migrate} + T_{control}, \tag{2}$$

where $T^b_{a-migrate}$ refers to the migration time of agent B and $T_{control}$ is the latency for delivery of a control message between agent B and A. Consequently, the suspend operation of agent A can be finished within the time of $t_{sus\_res} - t^a_{begin}$. Since B's migration is overlapped with A, the cost for

suspend of agent A can be approximated as

$$T^a_{suspend} = T_{control} + T^b_{suspend} + \tau. \quad (3)$$

Non-overlapped concurrent migration. In this case, we assume agent A issues a suspend request earlier than agent B's. According the timing sequence in Figure 4(b), A's request gets confirmed and hence its suspend operation takes the same time as in the single migration pattern. For agent B, its suspend operation won't be issued until a RESUME message from agent A is received. The waiting time is equal to $T^a_{suspend} + T^a_{a-migrate} + T_{control} + \tau$. In fact, B's waiting time is overlapped with A's migration. As far as connection migration is concerned, B saves the cost for suspend operation. Hence, we have

$$T^b_{c-migrate} = T_{resume} + T_{control} + \tau. \quad (4)$$

5.2 Simulation Results

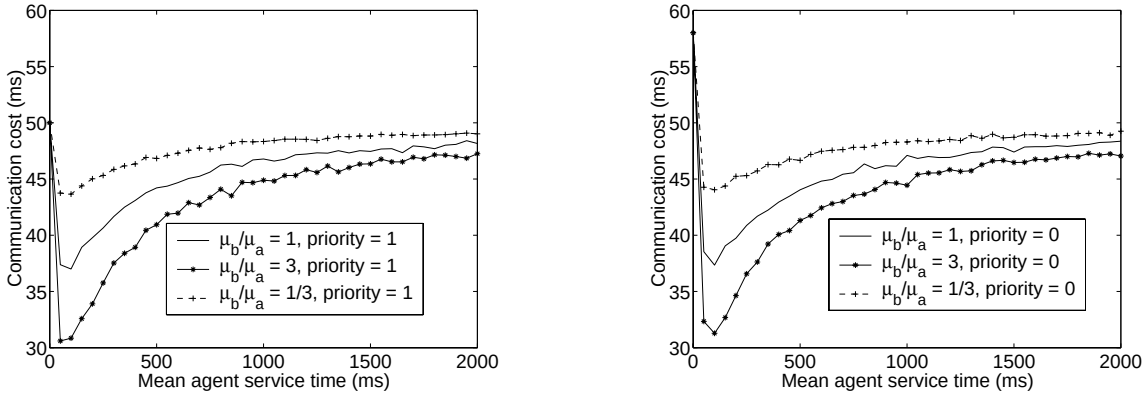
The performance model in the preceding section reveals that the cost for connection migration $T_{c-migrate}$ depends on the suspend starting time t^a_{begin} and t^b_{begin} and their interval τ, in addition to the cost for delivery of a control message ($T_{control}$) and the cost for suspend and resume operations ($T_{suspend}$) and (T_{resume}). The starting time T_{begin} is in turn determined by the agent migration pattern, characterized by migration frequency.

In this simulation, we set $T_{control}$, $T_{suspend}$, and T_{resume} to 10ms, 27.8ms, and 16.9ms, respectively, as we measured in our experiments in Section 4.2. In addition, we set the cost for agent migration $T_{a-migrate}$ to be 220ms. We evaluated the impact of migration frequency by modeling it as a random variable. We assumed the random variable follows an exponential distribution with an expectation of μ.

Figures 12(a) and 12(b) show the cost for connection migration of NapletSocket with the change of mean service time for agent A (i.e., $1/\mu^a$). Plots for different service time for agent B $1/\mu^b$, relative to $1/\mu^a$, are also presented. When two agents migrate with a very high speed (i.e. small service time), there are more chances for concurrent connection migrations. When they migrate with a low speed (i.e. large service time), a single connection migration most likely occurs. In both cases, the cost for connection migration, $T_{c-migrate}$, remains unchanged for the high priority agent. By contrast, the low priority agent experiences a little more delay when both of the agents migrate at a high speed.

From Figure 12, we can also see that the lowest latency for both high and low priority agents happens around the point where their starting time interval τ is larger than $T_{control}$, but not large enough to become single migration patterns. The plots with different settings of μ^b/μ^a also show that given an A's migration rate, increase of the ratio means agent B migrates faster so that when agent A suspends a connection, it has more chances of meeting an ongoing suspend request from B. This leads to a block of agent A's suspend requests and the overall cost for agent A's connection migration gets decreased.

Finally, we examine the impact of message exchange rate on the cost for connection migration. Instead of measuring the cost by time, this simulation uses the metric of number of control messages



(a) Cost for high priority agent.

(b) Cost for low priority agent.

Figure 12: Connection migration cost of NapletSocket during agent migration.

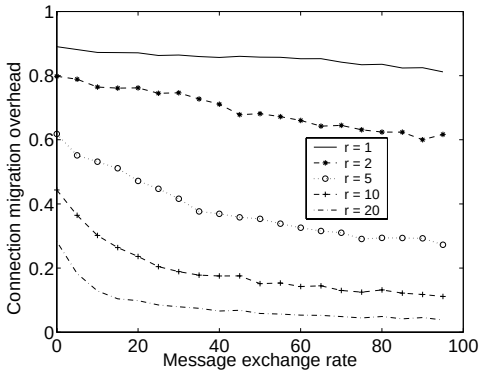


Figure 13: Connection migration overhead of NapletSocket with agent communication

involved in each connection migration, relative to the number of data messages communicated through the established connection. The message exchange rate, denoted by λ, refers to the number of data messages transmitted in a time unit. We define $r = \lambda/\mu$ as a relative message exchange rate, with respect to migration frequency μ. Figure 13 shows the connection migration overhead with different combinations of message exchange rates and migration frequencies.

From the figure, we can see that for a fixed ratio *r*, when the message exchange rate is small, the agent issues relatively more control messages to maintain a persistent connection and hence more overhead incurs. As the message exchange rate increases, the overhead is amortized over each communication. When the ratio r decreases to as low as one, which means the agent communicates once in each host, the overhead for persistent connection is always above 80% no matter how large the message exchange rate is.

6 Related Work

Communication between mobile agents has long been an active research topic. Agent communication languages like KQML [7] and FIPA's ACL [8] are designed for interactions between autonomous agents that originate in different places. In literature, there also exist many protocols in support of asynchronous communication between mobile agents; see [19] for a recent comprehensive review. A widely-used approach is mailbox-based mechanism [2], in which each agent is associated with a mail-box. A message is either sent directly or forwarded to the mail-box before the receiver agent retrieves it. In [3], the authors furthered the mailbox mechanism for reliable communication. It is achieved by performing synchronization between delivery and mailbox's migration. For agent that involves long migration path, synchronization with mail-box may become a bottleneck. Their work is also characterized by fault-tolerant support in case of network and host failure.

Another example of reliable asynchronous persistent communication mechanisms is due to Murphy, *et al.*[14]. They proposed a solution for message delivery to highly mobile agent based on the well known distributed global snapshot algorithm. They made it equivalent for message delivery to an agent and messages recording in distributed snapshot. However, their work is limited by its FIFO communication assumption, which may not always be the case.

We note that asynchronous persistent communication is not sufficient for applications that require agents to cooperate closely. A synchronous transient communication would keep the agents working more closely and efficiently. Mishra, *et al.* [13] proposed comprehensive interagent communication models for mobile-agent systems. Their proposed synchronous location-independent communication is similar to ours in application semantics. But they achieved the communication by a centralized clearinghouse, with which send/receive operations are matched and addresses of each other are returned. After that the sender sends the message directly to the receiver. This has a large message delivery latency since it requires at least twice the one-way message delay plus processing time.

In the research field of connection migration, there are a number of techniques proposed in different contexts.The conventional technique is from network-layer, using the same IP address even when users change network attachment point. An example of this technique is Mobile IP [11] which works on a concept of home agent associated with the mobile host. Every package destined to a mobile host by its home address is intercepted by its home agent and forwarded to it.

Network layer implementations are not appropriate or sufficient for some applications [1]. There are many studies focusing on transport layer support for mobility. One of the representative work is due to Snoeren [17, 1]. It uses an end-to-end mechanism to handle host mobility, by extending the TCP protocol with a TCP migrate option. The semantics of TCP remains unchanged. There are other similar work, such as TCP-R [9], M-TCP [18]. Although most of them work well, they require to change OS kernels. This hinders the protocols from wide deployment.

Connection migration in network and transport layers is mainly for the support of physical mobility. Connection migration due to code mobility requires support at session layer. Zhang and Dao introduced a persistent connections model [24]. They described connection end points in terms of location-independent IDs, which are stored in a centralized host. When an end point changes its attach-

ment point, it notifies the host and then the host notifies all others in the system. By using a centralized host, their approach may suffer from poor performance.

Qu, *et al.* later proposed a similar scheme to preserve upper layer unbroken connections [16], using some OS-specific kernel interfaces to access the system buffer for data not yet delivered. Okoshi, *et al.* presented a library solution called MobileSocket on top of Java Socket [15]. They used dynamic socket switch to update connection of MobileSocket and application layer window to keep all in-flight data at user level so that data can be recovered from broken connections.

Similar mechanisms were used for robust TCP connections due to mask server crash or communication failures. Robust TCP connections [6] addresses reliable communication problem in the area of fault tolerant distributed computing. Reliable Sockets (Rocks) [23] allows TCP connections to support changes in attachment points with emphasis on reliability over mobility. It has support for automatic failure detection and a protocol for inter-operates with end points that do not support Rocks.

7 Conclusions

Mailbox-based asynchronous persistent communication mechanisms in mobile agent systems are not sufficient for certain distributed applications like parallel computing. Synchronous transient communication provides complementary services that make cooperative agents work more closely and efficiently. This paper presents a connection migration mechanism in support of synchronous communication between agents. It support concurrent migration of both agents in a connection and guarantees exactly-once message delivery. The mechanism uses agent-oriented access control and secret session keys to deal with security concerns arising in connection migration. A prototype of the mechanism, NapletSocket, has been developed in Naplet mobile agent system. Experimental results show that the NapletSocket incurs a moderate cost in connection setup, mainly due to security checking, and marginal overheads for communication over established connections. Furthermore, we investigate the impact of agent mobility on communication performance via simulation. Simulation results show that NapletSocket is effective and efficient for a wide range of migration and communication patterns.

Current work focuses on reliable problems caused by the presence of agent mobility. It has no support for detection and recovery from link or host failures. As part of on-going work, we are going to extend the NapletSocket for fault-tolerance.

References

[1] M. Alexander and C. Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, MIT, February 2003.

[2] J. Cao, X. Feng, J. Lu, and S. Das. Mailbox-based scheme for mobile agent communication. *IEEE Computer*, 35(9):54–60, 9 2002.

[3] J. Cao, L. Zhang, J. Yang, and S. Das. A reliable mobile agent communication protocol. In *Proc. 24th Int'l Conf. on Distributed Computing Systems*, March 2004.

[4] Chesapeake Computer Consultants. Tools - Test TCP (TTCP). http://www.ccci.com/tools/ttcp/.

[5] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[6] R. Ekwall, P. Urb'an, and A. Schiper. Robust TCP connections for fault tolerant computing. In *Proc. Int'l Conf. on Parallel and Distributed Systems*, 2002.

[7] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the 3rd Int'l Conf. on Information and Knowledge Management (CIKM'94)*. ACM Press.

[8] FIPA. FIPA ACL message structure specification, foundation for intelligent physical agents, 2001. http://www.fipa.com/.

[9] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. In *Proc. IEEE Int'l Conf. on Network Protocols, pages 229-236*, 1997.

[10] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. *Mobile Agents: Motivations and State-of-the-Art Systems*. In Jeffrey M. Bradshaw, editor, Handbook of Agent Technology, AAAI/MIT Press, 2000.

[11] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based protocols for mobile internetworking. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications(SIGCOMM)*, April 2002.

[12] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *INFOCOM (3)*, pages 1037–1045, 1998.

[13] S. Mishra and P. Xie. Interagent communication and synchronization support in the daagent mobile agent-based computing system. In *IEEE Transactions on Parallel and Distributed Systems, VOL. 14, NO. 3*, March 2003.

[14] A. L. Murphy and G. P. Picco. Reliable communication for highly mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*.

[15] T. Okoshi, M. Mochizuki, and Y. Tobe. Mobilesocket: Toward continuous operation for java applications. In *Int'l Conf. on Computer Communications and Networks, pages 50-57*.

[16] X. Qu, J. X. Yu, and R. P. Brent. A Mobile TCP Socket. In *Proc. of IASTED Int'l Conf. on Software Engineering*, November 1997.

[17] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th Int'l Conf. on Mobile Computing and Networking (MobiCom)*, 2000.

[18] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *Proc. 22nd Int'l Conf. on Distributed Computing Systems*, 2002.

[19] P. T. Wojciechowski. Algorithms for location-independent communication between mobile agents. In *Proceedings of AISB '01 Symposium on Software Mobility and Adaptive Behaviour*, pages 10–19, March 2001.

[20] C.-Z. Xu. Naplet: A flexible mobile agent framework for network-centric applications. In *Proc. of the Second Workshop on Internet Computing and also available at http://www.ece.eng.wayne.edu/ czxu/paper/icec02-naplet.pdf e-Commerce (In conjunction with IPDPS)*, April 2002.

[21] C.-Z. Xu and S. Fu. Privilege delegation and agent-oriented access control in naplet. In *Proc. of Int'l Workshop on Mobile Distributed Computing (In conjunction with ICDCS)*, April 2003.

[22] C.-Z. Xu and B. Wims. Mobile agent based push methodology for global parallel computing. *Concurrency: Practice and Experience*, 14(8):705–726, July 2000.

[23] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proc. 8th Annual ACM/IEEE Int'l Conf. on Mobile Computing and Networking, pages 95-106*.

[24] Y. Zhang and S. Dao. A persistent connection model for mobile and distributed systems. In *Int'l Conf. on Computer Communications and Networks*.