

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 become a promising technology to provide scalable services. Due to their mobile nature, it is a challenge to support 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 of 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 NapletSocket incurs a moderate cost in connection setup, mainly due to security checking, and marginal overhead for communication over established connections. Furthermore, we investigate the impact of agent mobility on communication performance via simulation. Simulation results show that NapletSocket is efficient for a wide range of migration and communication patterns.

1 Introduction

Mobile codes refer to programs that function as they are transferred from one machine to another. The concept of code mobility represents a promising solution for today's network services for load balancing, fault resilience, system administration, etc. A special form of migration is mobile agent that has the ability to travel autonomously, carrying its code, as well as data and running state. Because of its unique properties, mobile agents have been the focus of much speculation in the past decade.

In mobile agents based computing, it is often necessary for remote agents to communicate with each other to work efficiently. Conventional technologies for remote

inter-agent communication is through a mailbox-like *asynchronous persistent* communication mechanism due to the requirement for agent autonomy [1]. That is, an agent can send messages to others no matter its communication parties exist or not. Asynchronous persistent communication plays a key role in many distributed applications and is widely supported by existing mobile agent systems; see [13] for a comprehensive review of location independent communication protocols between mobile agents.

A common asynchronous communication mechanism works in 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. This communication model doesn't guarantee instantaneous message delivery and it is hard for the sender to determine whether and when the receiver gets the message. Thus it is not sufficient for applications that require agents to closely cooperate. For example, in the use of mobile agents for parallel computing [16], 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 synchronous 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 support 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 [6, 7, 8, 11, 12]. We refer to this type of mobility as physical mobility, in contrast to logical mobility of codes. Although these protocols provide feasible ways to link mobile devices to network, they have no control over the logical mobility.

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 [9, 10, 17, 18]. However, none of them was targeted at agent mobility. Agent related connection migration involves two unique reliability and security problems. Reliable agent connection migration needs to have mobility support for exactly-once delivery of all transmitted data, even if the two agents migrate simultaneously. Security is a major concern in agent-oriented programming. Socket is a critical resource and its access must be fully controlled by agent servers. Connection migration is vulnerable to eavesdropper attacks and it is necessary to protect transactions over a connection from any malicious attacks.

In this paper, we present the design and implementation of an integrated mechanism that deals with reliability and security in agent connection migration. It provides an agent-oriented socket programming interface for location-independent socket communication and guarantees exactly-once message delivery. To assure secure connection migration, each connection 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 [14] 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.

The remainder of the paper is organized as follows. Section 2 and Section 3 give an overview and details of the design of NapletSocket. Section 4 presents experimental results. Section 5 presents a communication performance model and simulation results on the impact of agent mobility. Related work is summarized in Section 6. Section 7 concludes the paper.

2 NapletSocket: A Connection Migration Mechanism

2.1 NapletSocket Architecture

NapletSocket provides interfaces 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 NapletSocket connection is agent oriented. It is known 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. The 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

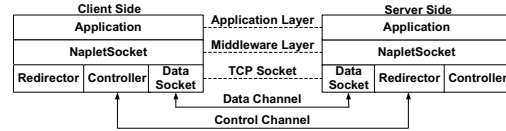


Figure 1. NapletSocket Architecture.

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.

established, all communications are through the connection and location service is no longer needed.

To support connection migration, NapletSocket 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 NapletSocket architecture. It comprises of three main components: data socket, controller and redirector. The data socket is the actual channel for data transfer. It is associated with an input buffer to keep undelivered data. The controller is used for management of connections and operations that need access rights to socket resources. The redirector is used to redirect socket connection from a remote agent to a local resident agent. Both the controller and the redirector can be shared by all NapletSockets so that only one pair is necessary in a Naplet server.

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. In each state, certain action will be taken when an appropriate event occurs. There are two types of events: calls from local agents and messages from remote agents. Actions include sending messages to remote agents and invoking 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 servers. Details of the open, 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

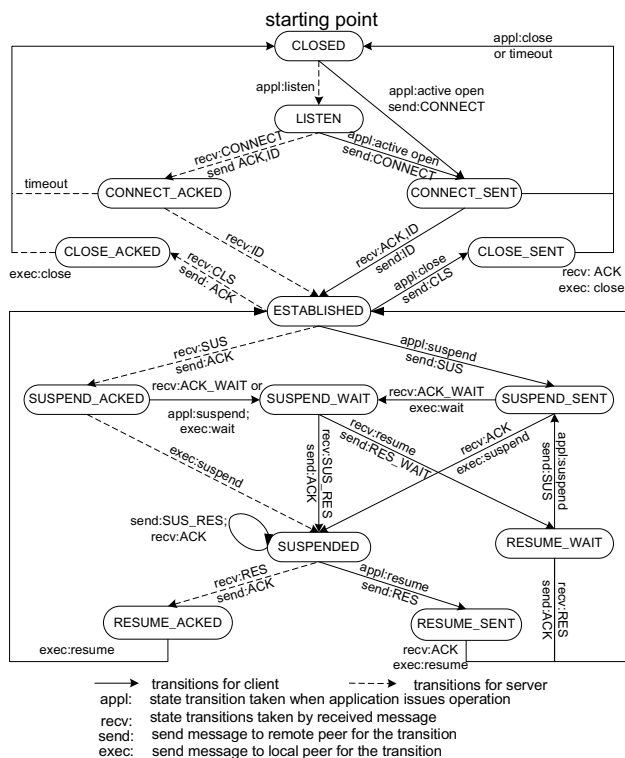


Figure 3. State transitions diagram.

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. The connection then changes to the CONNECT_ACKED state. After the socket ID of the client side is received, the state switches to ESTABLISHED. Now data can be transferred between the two peers as normal socket connection.

Suspend/Resume a connection. Either of the two peers may suspend an established connection. When invoked, the suspend operation sends a SUS to the other side. If the request is acknowledged, an ACK is sent back and triggers the action of closing underlying I/O 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 connection state for this peer changes to SUSPENDED. No data can be exchanged in this state.

In the SUSPENDED state, when either of the agents decides to resume the connection, it invokes the resume inter-

face. 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 to ESTABLISHED. Once the remote peer in the SUSPENDED state receives a resume request, it first sends back an ACK. Then the redirector hands its connection to the desired NapletSocket and new I/O streams are created. After that, the connection changes to ESTABLISHED.

Close a connection. In either the ESTABLISHED or the SUSPENDED state, the close interface can be invoked to close the current connection. A CLS(CLOSE) request is sent to the peer. After acknowledgement, both sides may close the underlying socket and I/O streams. Then the connection state changes to CLOSED.

3 Design Issues

3.1 Transparency and Reliability

Connection migration needs to be transparent to end users. The main approach is to use a data socket under NapletSocket. During connection migration, the underlying data socket is first closed before migration and updated afterward. It is possible there are in-flight data at the time of migration. To guarantee messages delivery, we add an input buffer to each input stream and wrap them together as a NapletInputStream. To suspend a connection, the operation retrieves all currently undelivered data into the buffer before closing the data socket. The data 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. It will only read data from the new socket stream after all data in the buffer have been retrieved.

When two agents of a connection move simultaneously, there is a problem since the resume operation only remembers the previous destination. It is not difficult to get information of the new destination. The problem is if an agent migrates frequently, a resume operation may have to chase the agent. To avoid the problem, we provide a simplified solution by delaying one of the migration, which means one agent migrates first and the other must wait for the first one to finish. Therefore agents migrations occur sequentially although they are issued concurrently. But from the viewpoint of high level applications, the underlying sequential migration is transparent.

To delay one of the two simultaneous suspend requests, two states SUSPEND_WAIT and RESUME_WAIT are used. One of the suspend operation is delayed and the connection is put into the SUSPEND_WAIT state. At this state, the operation is blocked until the other agent finishes migration and sends a SUS_RES message. RESUME_WAIT is the state when a resume operation is blocked. After the first migration, the resume operation is invoked and the underlying socket is updated. But the connection needs to be suspended due to

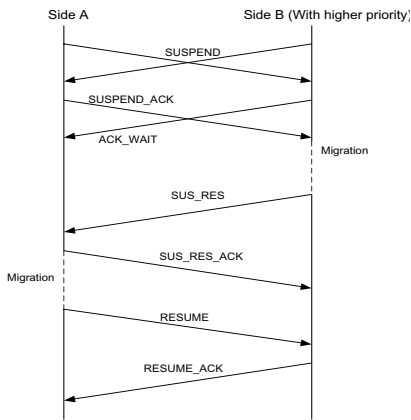


Figure 4. Sequence of overlapped migration.

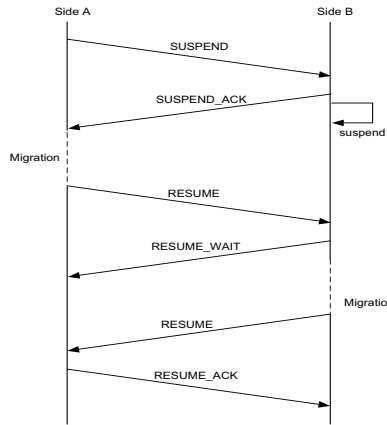


Figure 5. Sequence of non-overlapped migration.

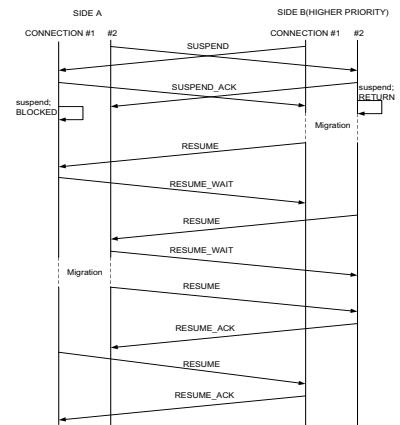


Figure 6. Sequence of multiple connections migration.

the second agent migration and the underlying socket may have to be closed. To prevent this extra update, we block the resume operation and change the state of the connection to RESUME_WAIT. The resume operation will get signaled after the second migration and finish.

Depending on the ordering of two suspend operations, we classify the problem into two types. One is overlapped concurrent connection migration; the other is non-overlapped. In the first case, both SUSPEND requests are issued simultaneously and neither side receives an acknowledgement before the SUSPEND request. Both sides have to decide whether to approve the request. To approve one and only one request, we give higher priority to one of the agents. The priority is decided by unique agent IDs. An agent with a larger ID has a higher priority. A time sequence for this case is in Figure 4, with agent in side B having a high priority. Side B receives a SUSPEND request and since it has also sent a request, it knows this is a concurrent connection migration. Then it sends back an ACK_WAIT to delay the migration. Side A has the same situation. But side A always acknowledges a SUSPEND request since it has a low priority. After that, the state of the connection in side B switches to SUSPENDED and in side A it switches to SUSPEND_WAIT. The agent in side B migrates and informs side A with a SUS_RES message. Side A can then proceed with normal connection suspend and resume.

In the second case, agent in one side issues a suspend request after a SUSPEND request is acknowledged and the request processing hasn't finished. So no new suspend operation should be issued until the first one finishes. A time sequence is presented in Figure 5 as an example.

3.2 Multiple Connections

In the preceding discussions on concurrent connection migration, we focused on one connection. When multiple connections are established, all connections should be sus-

pending before agents migration. During the suspend operations, it is possible for them to be suspended in different orders. For example, suppose there are two connections by agents in side A and B, represented as #1 and #2. Side A may suspend the connections in the order of #2, #1 and side B in the order of #1, #2. During concurrent migration, it is possible for side A to suspend connection #2 while at the same time side B tries to suspend #1. Neither these operations needs to be delayed because they are operating on different connections. Thus both connections are successfully suspended. When suspending the second connection, side A on #1 while side B on #2, both sides will find the connections already suspended. By default a suspend operation simply returns for a suspended connection. Therefore both sides successfully suspend their connections and agents migrations happen at the same time. As a result, neither knows where the other agent is after migration.

To ensure only one agent migrates at a time, we give different responses to a suspend operation when it is operating on a suspended connection. Suspend operations are distinguished by whether they are issued locally, *suspend locally*, or invoked by remote messages, *suspend remotely*. In a suspend operation, if the connection has already been suspended remotely which means there is an on-going agent migration in the other side, we decide whether to continue or block depending on the priority of the agent. If it has a low priority, we block the suspend operation; if it has a high priority, we finish the operation without further actions. The blocked suspend operation will get signaled when the one with high priority finishes migration. Figure 6 gives an illustration of the protocol in the case of two connections.

3.3 Security

Security is always a major 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. Second, the connection should be secure from possible attacks like eavesdropping. More specifically, a connection can only be suspended/resumed/closed 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. This can be achieved 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. An agent subject has no permission to access local socket resources. When it needs such access, it submits a request to the controller. The controller authenticates the agent and checks access permissions. If the security check passes, a NapletSocket or NapletServerSocket is created and returned to the agent. More details about agent-oriented access control in Naplet system can be found in [15].

Regarding the second issue, connection migration can be realized by the use of a socket ID. However, a plain socket ID couldn't prevent eavesdropping attacks. To this end, we apply Diffie-Hellman key exchange protocol to establish a secret session key between the pair of communicating agents at connection setup stage. Any subsequent operations on the connection must be accompanied with the secret key. Such requests will be denied unless their keys are verified. This protects NapletSocket connections from eavesdropping attacks.

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. Then we evaluate the overall communication performance in case of agent migration.

4.1 Effectiveness of Reliable Communication

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

Figure 7 shows a trace of the message counters received by the mobile agent at different time. Agent migration time is omitted for simplicity. The agent migrates at 10th, 20th, 30th milliseconds. The light dots show the messages read from the socket stream and the dark dots are those into or

Table 1. Cost of open/close operations.

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

from the NapletSocket buffer. In the first migration point, the agent migrates before it retrieves all the messages in transmission. The undelivered three messages (7, 8, 9) are kept in the buffer. They are transferred together with the agent and delivered to the application after migration by the support of NapletSocket. Similarly, the third agent migration involves transferring of one message.

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.

We performed open and close operations with and without security checking for 100 times. Table 1 shows the average time for both operations. From the table, we can see that opening a secure mobile connection costs almost 40 times as much as that of a Java socket. A breakdown of the cost 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.

Similarly, we recorded 27.8ms and 16.9ms for suspend and resume operations, respectively. The costs are mainly due to the exchange of control messages, which makes up about 50% for suspend and 70% for resume.

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 afterwards, the total cost involved is 147ms. However, if we use suspend and resume instead, the cost is only 44.7ms. The total time saved increases with the number migration.

4.3 NapletSocket Throughput

In the third experiment, we tested NapletSocket throughput by the TTCP [3] measurement tool, in which a pair of TTCP programs is used to communicate messages of different sizes as fast as possible. Figure 8 shows NapletSocket throughput between two stationary agents. The throughput of Java Socket is included for comparison. From the figure, we can see the NapletSocket throughput degrades slightly. This degradation is mainly due to synchronized access to I/O streams. With the increase of message sizes, the performance gap becomes almost negligible.

We then measured NapletSocket throughput with different agent migration frequency (i.e. service time at each hop). We used a *single migration* pattern where one agent

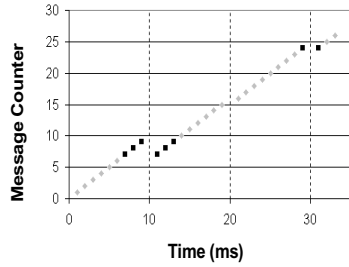


Figure 7. A trace of message transmissions and deliveries.

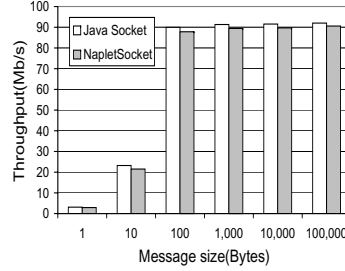


Figure 8. Throughput of NapletSocket versus Java Socket.

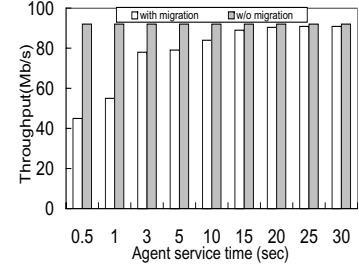


Figure 9. Impact of migration frequency on effective throughput.

remains stationary while the other keeps moving at a certain rate. We refer to the total traffic communicated over a period of communication and migration time as *effective throughput*. In this experiment, we assume a constant message size of 2K bytes. Figure 9 shows the results. We can see the throughput is only about 45Mb/s when the service time is 0.5 second. If an agent stays in a host long enough, for example 10 seconds, the effective throughput gets very close to the stationary throughput. This implies that the effect of agent and connection migrations on throughput becomes negligible when an agent migrates at a low frequency.

5 Performance Model of Agent Mobility

The experiments in the preceding section assumed agents communicate to each other “as fast as possible”. The objective of this section is to investigate the impact of the communication rate as well as the agent migration concurrency on the performance of NapletSocket.

5.1 Performance Model

Consider two mobile agents, say A and B, which are connected via a NapletSocket connection. Both agents travel around the network at various rates. Without loss of generality, we assume agent B has a higher priority. At each host, the agents process their tasks for certain time and communicate with each other for synchronization. Then the agents migrate to other hosts. Associated with each agent migration is a connection migration, with a suspend operation before agent migration and a resume operation afterward.

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

Notice that the costs for suspend and resume operations are related to agent migration concurrency. In the case of

single migration, both $T_{suspend}$ and T_{resume} are constant. When both endpoints of a connection are mobile, both costs are dependent upon the ordering of their requests. Let t_{begin}^a and t_{begin}^b denote the request time of the two agents A and B and their interval $\tau = |t_{begin}^a - t_{begin}^b|$. If τ is large enough for the first suspend to complete before the second request, it is single migration. Otherwise, it is concurrent migration.

As we discussed in Section 3.1, concurrent migration can be further distinguished by being overlapped or non-overlapped. In the overlapped case, the suspend cost of agent B, $T_{suspend}^b$, is the same as that of a single migration, since it has a higher priority. The suspend operation of agent A couldn't continue until B finishes and sends a SUS_RES message, as shown in Figure 4. From the figure, we can see that the arrival time of SUS_RES at agent A $t_{sus.res} = t_{begin}^b + T_{suspend}^b + T_{a-migrate}^b + T_{control}$, where $T_{a-migrate}^b$ refers to the migration time of agent B and $T_{control}$ is the latency for delivery of a control message between agent A and B. Consequently, the suspend operation of agent A can be finished within the time of $t_{sus.res} - t_{begin}^a$. Since B's migration is overlapped with A, the suspend cost for connection of agent A can be approximated as $T_{control} + T_{suspend}^b + \tau$.

In the non-overlapped case, as shown in Figure 5, agent A issues a suspend request earlier than B. A's request gets confirmed and its suspend operation takes the same time as in the single migration. The suspend operation of agent B won't be issued until a RESUME message from agent A is received. The waiting time is equal to $T_{suspend}^a + T_{a-migrate}^a + T_{control} + \tau$. In fact, B's waiting time is overlapped with A's migration. As far as connection migration is concerned, B saves the cost for a suspend operation. Hence, we have $T_{c-migrate}^b = T_{resume} + T_{control} + \tau$.

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_{begin}^a and t_{begin}^b and their interval τ , in addition to the cost for delivery of a control message $T_{control}$ and the cost for suspend and resume opera-

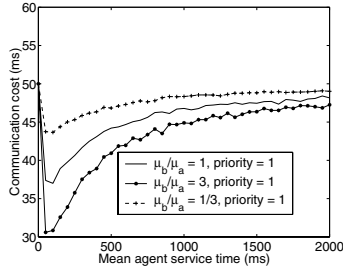


Figure 10. Connection migration cost for high priority agent.

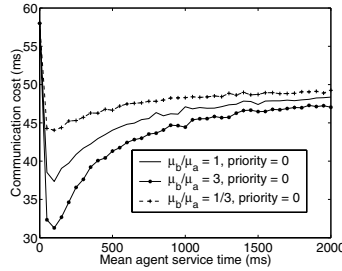


Figure 11. Connection migration cost for low priority agent.

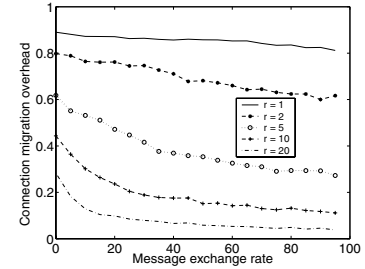


Figure 12. Connection migration overhead with agent communication

tions $T_{suspend}$ 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} as 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 $T_{a-migrate}$ as 220ms, measured in Naplet system. We evaluate the impact of migration frequency by modeling it as a random variable following exponential distribution with expectation μ .

Figures 10 and 11 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 agents migrate at a high speed.

From the figures, we can also see that the lowest latency for both agents happens around the point where τ is larger than $T_{control}$, but not large enough for a single migration. Given agent A's migration rate, increase of the ratio μ^b/μ^a 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 its connection migration gets decreased.

Finally, we examine the impact of message exchange rate on the cost for connection migration in terms of the number of control messages relative to the number of data messages communicated. 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. Figure 12 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 [4] and FIPA's ACL [5] focus on a semantic level of agent communication. In literature, there also exist many protocols in support of asynchronous communication between mobile agents; see [13] for a recent review. A widely-used approach is mailbox-based mechanism [1], in which each agent is associated with a mail-box. A message is either sent directly or forwarded to the mail-box before delivered. In [2], the authors furthered the mailbox mechanism for reliable communication. It is achieved by performing synchronization between delivery and mailbox's migration. This forwarding based mechanism is not suitable for applications that need instantaneous message delivery. Furthermore, if agent involves long migration path, synchronization with mail-box may become a bottleneck.

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 [7] 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 many applications. There are many studies focusing on transport layer support for mobility. One of the representative work is [11], which 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 is other similar work, such as TCP-R [6], M-TCP [12]. Although most of them work well, they require to change OS kernels. This hinders the protocols from wide deployment.

There are some session-layer connection migration mechanisms. In [18], the authors introduced a persistent connections model. They described connection end points in terms of location-independent IDs, which are stored in a centralized host. When an end point changes its attachment point, it notifies the host and then the host notifies all others. In [10], the authors proposed a scheme to preserve upper layer unbroken connections using some OS-specific kernel interfaces to access the system buffer for data not yet delivered. In [9], the authors presented a library solution called MobileSocket on top of Java Socket. 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. Similarly, in [17], the authors proposed Reliable Sockets (Rocks) that 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-operation 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 NapletSocket incurs a moderate cost in connection setup and marginal overhead 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.

Acknowledgement

This research was supported in part by NSF grants CCR-9988266 and ACI-0203592.

References

- [1] J. Cao, X. Feng, J. Lu, and S. Das. Mailbox-based scheme for mobile agent communication. *IEEE Computer*, 35(9):54–60, 2002.
- [2] 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.
- [3] Chesapeake Computer Consultants. Tools - Test TCP (TTCP). <http://www.ccci.com/tools/ttcp/>.
- [4] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Software Agents*. AAAI/MIT Press, 1997.
- [5] FIPA. FIPA ACL message structure specification, foundation for intelligent physical agents, 2001. <http://www.fipa.org/>.
- [6] 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, October 1997.
- [7] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based protocols for mobile internetworking. In *Proc. ACM SIGCOMM*, pages 235–245, September 1991.
- [8] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom*, pages 1037–1045, September 1998.
- [9] T. Okoshi, M. Mochizuki, and Y. Tobe. Mobilesocket: Toward continuous operation for java applications. In *Proc. Int'l Conf. on Computer Communications and Networks*, October 1999.
- [10] X. Qu, J. X. Yu, and R. P. Brent. A mobile TCP socket. In *Proc. IASTED Int'l Conf. on Software Engineering*, November 1997.
- [11] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM MobiCom*, August 2000.
- [12] 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*, July 2002.
- [13] P. T. Wojciechowski. Algorithms for location-independent communication between mobile agents. In *Proc. AISB '01 Symposium on Software Mobility and Adaptive Behaviour*, pages 10–19, March 2001.
- [14] C.-Z. Xu. Naplet: A flexible mobile agent framework for network-centric applications. In *Proc. of the 2nd Workshop on Internet Computing and e-Commerce (In conjunction with IPDPS)*, April 2002.
- [15] 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.
- [16] 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.
- [17] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proc. ACM MobiCom*, September 2002.
- [18] Y. Zhang and S. Dao. A persistent connection model for mobile and distributed systems. In *Proc. Int'l Conf. on Computer Communications and Networks*, September 1995.