A Protocol Design of Communication State Transfer for Distributed Computing *

Kasidit Chanchio Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{kasidit, sun}@cs.iit.edu

Abstract

This paper presents the design of a communication state transfer protocol to support process migration in a dynamic, distributed computing environment. In our design, processes in distributed computation communicate one another via message passing and are migration-enabled. Due to mobility, mechanisms to maintain reliability and correctness of data communication are needed. Following an event-based approach, such mechanisms are derived to handle various communication situations when a process migrates. These mechanisms collectively preserve the semantics of the communication and support efficient communication state transfer.

1 Introduction

Process migration is a basic function of dynamic programming. It moves a running process from one computer to another. Motivations of process migration include load balancing, fault tolerance, data access locality, resource sharing, reconfigurable computing, and system administration, etc [1]. Recent research shows process migration is necessary for achieving high performance via utilizing unused network resources. Process migration is a fundamental technique needed for the next generation of internet computation [2]. However, despite these advantages, process migration has not been adopted in engineering practice due to its design and implementation complexities, especially in a large distributed environment.

Snow [3] is a distributed environment supporting user-level process migration. Snow provides solutions for three problem domains for transferring computation state, memory state, and communication state of a process. Methods to transfer the execution and memory state have been developed [3]. This paper presents a solution to transfer the communication state.

Activities in a large-scale distributed environment are dynamic in nature. Adding process migration functionality makes data communication even more challenging. A number of fundamental problems have to be addressed. First, processes should be able to communicate one another from anywhere and at anytime. Process migration could occur during a communication. Mechanisms need to be developed to guarantee correct message arrivals. Second, the problem of updating location information of a migrating process has to be solved. After a process migrates, other processes have to know its new location for future communications. The updating technique should be scalable enough to apply to a large network environment. Finally, if a sequence of messages are sent to a migrating process, correct message ordering must be maintained.

We have developed data communication and process migration protocols working cooperatively to solve the aforementioned problems. This paper focuses on basic ideas behind our protocol design and discusses how the protocols work under various communication situations. Our protocol design is based on the concept of point-to-point connection-oriented communication. It aims to provide a robust and general solution for communication state transfer. Mechanisms to handle process state transfer are implanted into a number of communication operations which could occur at data communication end points. These operations include send and receive operations in the data communication protocol and migration operation in the process migration protocol. They coordinate one another during the migration to guarantee correct message passing. The protocols are naturally suitable for large-scale distributed environment due to their inherited properties. First, they are scalable. During a migration, the protocols coordinate only those processes di-

*This work was supported in part by National Science Foundation under NSF grant ASC-9720215, CCR-9972251, and by IIT under the ERIF award.

rectly connected to a migrating process. The migration operations are performed mostly at the migrating process, while communication peer processes are only interrupted shortly for the coordination. In our design, residual dependency disappears as soon as the coordination finishes and all existing communication connections on a migrating process are closed down. Second, the process migration protocol is non-blocking i.e., it allows other processes to send messages to the migrating process during the migration. These two properties are quite beneficial for large environments where the number of participating processes is high. Third, the protocols do not create deadlock. They prevent circular wait, while coordinating a migrating process and its peers for migration. Finally, the protocols are simple in implementation and are practical for wide-area, networking environments. They can be implemented on top of existing connection-oriented communication protocols such as PVM (direct communication mode) and TCP.

2 Related Work

Mechanisms to support correct data communication can be classified into two different approaches. The first approach is using existing fault-tolerant, consistent checkpointing techniques. To migrate a process, users can "crash" a process intentionally and restart the process from its last checkpoint on a new machine. Since global consistency is provided by the checkpointing protocol, safe data communication is guaranteed. Projects such as CoCheck [1] follow this approach.

On the other hand, mechanisms to maintain safe data communication can be implemented directly into data communication and process migration protocols. When a process migrates, process migration operations coordinate with data communication operations on other processes for reliability. Unlike the previous approach, the protocols only need to maintain communication consistency between the migrating process and its peers. Therefore, their designs are more scalable and less costly. Snow, Charlotte [1], and MPVM [4] are in this direction. These projects are message-based and rely on the concept of communication channel. Although sharing certain similarity, they have different designs. Charlotte implements reliable communication mechanisms in kernel and supports nonblocking communication. Although allowing to write highly concurrent programs, the communication mechanisms in Charlotte is complex. On the other hand, Snow and MPVM implement message passing mechanisms at user-level and support blocking point-to-point communication. Both protocols are implemented on top of PVM. The major differences between our work and MPVM are in the issues of process identification and communication re-establishment. From our experience, Snow's protocols demonstrate simple, yet efficient implementations on top of existing communication software. Further comparisons can be found in [5].

3 Protocol Designs

We consider a distributed computation as a set of collaborative processes executing under a virtual machine environment. Processes communicate one another by passing messages via FIFO communication channels. The virtual machine environment is a collection of software and hardware to support distributed computations. It has three basic components. First, a network of workstations is the basic resource for process execution. Second, a number of daemon processes residing on workstations comprise a virtual machine providing resource accesses and management. Finally, a scheduler is built to supervise resource utilization. Unlike in static distributed environments such as that supported by PVM and MPI, the scheduler is a necessary component of a dynamic distributed environment such as the Grid [2]. In our design, a process is identified by a *rank* number, a non-negative integer. Since the rank number is location transparent, location information must be maintained internally by every process and the scheduler. When a process migrates, the location information must be updated accordingly.

Based on connection-oriented communication, we define the communication state of a process to include all communication connections and messages in transit. To migrate the communication state, one has to capture the state information, transfer it to a destination computer, and successfully restore it there.

Migrating a communication state is non-trivial since various communication situations can occur during the migration. For better understanding, we define a process to generate a sequence of events [6]. An event is defined as an encapsulation of operations to perform certain function. We are interested in the computation, send, receive, and migration events. Supposed that a *MP* is a location in the space-time diagram where a processor migrates, Figure 1 gives example communication situations during a migration of $P2$. There are four concurrent processes in the example. Based on the connection-oriented concept, a connection is established between $P1$ and $P2$ when $m1$ is sent. As a result, the same communication channel is used to transmit $m2$ and $m3$. When the migration occurs at *MP*, there are three communication situations to be considered.

1. We need mechanisms to handle messages in transit during process migration. From the figure, this is the case for the transmissions of $m2$ and $m3$.

2. Mechanisms to handle message transmissions from unconnected processes during the migration must be considered. This is the case of $m4$.

3. Finally, we need mechanisms to handle message transmissions after process migration completes. This situation covers the transmissions of $m5$ and $m6$.

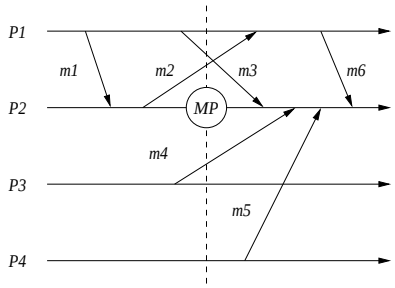

Figure 1. Communication situations.

Capturing and transferring messages in transit

In the first case, to capture messages in transit, processes on both ends of a communication channel have to coordinate each other to receive the messages until none is left. Then, the processes can close the empty channel. Note that a process is free to migrate only after all of its existing channels are closed down. We present the following schemes for the coordination:

A1. After receiving a migration instruction from the scheduler, the migrating process coordinates its connected peers by sending *marker* messages. The marker is also the last message. Since the channel is FIFO, the reception of the marker on the peers implies other messages sent prior to the marker have already been received. Upon receiving the markers, the peers send back acknowledgements which also indicate the last messages from the peers. After sending the markers, the migrating process reads messages out of the channel until the acknowledgement is received. Then, the channel is free for closure. From the example, $P2$ would initiate such coordination with $P1$. This coordination mechanism is based on the work of Chandy and Lamport [6].

A2. For efficiency, messages have to be drained from the channel as soon as possible and must be stored somewhere for future uses. A memory storage, namely the *received-message-list*, in user space of every process is needed for such purpose. From the examples, $m2$ and $m3$ are kept in the received-message-list of $P1$ and $P2$, respectively.

A3. Because messages may be stored in the receive-message-list before needed, the receive operation has to search for a wanted message from the list before taking a new message from a communication channel. Also, the new messages would be appended to the list until the wanted message is found.

After the coordination, messages in transit are captured and existing communication connections are closed down.

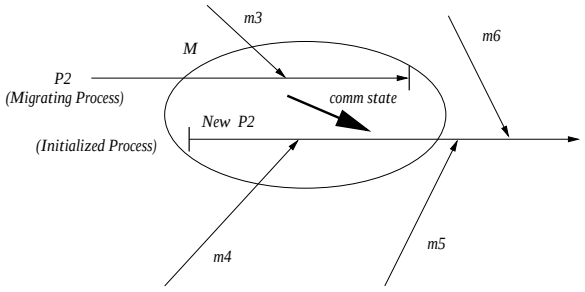

Figure 2. Activities in the migration event M.

One may consider the messages stored in the received-message-list of the migrating process as a part of the process's communication state which has to be transferred to the destination computer. Based on the previous example, Figure 2 illustrates a possible situation where a migration event M is operated at the migration location *MP*. In order to capture the transmitting messages, $P2$ coordinates $P1$ and receives $m3$ into the received-message-list which would be forwarded (as "*comm state*") to the process *New P2* on a destination computer. Note that *New P2* is an executable copy of $P2$ which is loaded on the destination of process migration to wait for process state transfer from the original $P2$. The loading is supervised by the scheduler and must be performed before the migration starts. Therefore, at the moment of migration, the scheduler knows exactly where $P2$ is going to be. Once the state transfer completes, the location information of $P2$ in the scheduler and $P2$ itself would be updated to the new one.

Migration-aware connection establishment

To handle data communication between unconnected processes, the connection establishment mechanisms have to be able to detect migration activities on the connecting processes and automatically resolve the problem. The following schemes are employed:

B1. Since our message passing operations only employ send and receive primitives and do not support explicit commands for connection establishment, the establishment mechanisms are installed inside the send and receive operations hidden from the application process.

B2. To establish connections, we employ the sender-initiated technique where the sender sends a connection request to its intended receiver process. Having process migration into the picture, the establishment mechanisms must be able to detect the migration (or past occurrence of the migration) and inform the sender process. In our design, the migration is detected once the sender receives a denial to its connection request. The rejection message could come either from the virtual machine or the migrating process. The virtual machine sends a rejection message in case the mi-

grating process has already been migrated. On the other hand, the migrating process rejects connection during migration operations. The migrating process starts migration operations when it receives a migration instruction from the scheduler and finishes the operations when process state transfer completes. In our migration protocol, the scheduler first decides a destination. After that, it loads the new process there and then instruct a process to migrate. Thus, the connection request is rejected only after the new process exists and the migration instruction has been intercepted.

B3. Once the migration is detected, the sender consults the scheduler to locate the receiver. After getting a new location, the sender updates the receiver's location, establishes a connection, and sends messages. Based on this scheme, in Figure 2, when $P3$ tries to send $m4$, the connection establishment mechanisms would detect the migration of $P2$, consult the scheduler, and redirect the establishment attempt to the process *New P2* instead. On the other hand, the connection establishment from $P4$ detects the past occurrence of process migration on the sending of $m5$ before redirecting its connection attempt to *New P2.* Similar situation occurs in the sending of $m6$ (since the connection between $P1$ and $P2$ has previously been closed down.)

Observation 1: Because of the process coordination and migration-aware connection establishment mechanisms, a sender process is not blocked while sending messages to a migrating process.

Observation 2: The updating of location of the migrating process is always performed by a sender process as a part of connection establishment. Since the establishment is performed "on demand" and does not need global synchronization, the updating mechanism is scalable.

Communication state restoration

The scheme for restoring of communication state on a new process can be addressed in two parts. First, contents of the receive-message-list forwarded from the migrating process are inserted to the front of the receive-message-list of the new process. This scheme restores the messages which are in transit during the migration. Second, messages sent from a newly connected process to the new process are appended to the end of the list. This scheme ensures message ordering. For Figure 2, the received-message-list of $P2$ would store $m3$ as a result of process coordination and then being forwarded to *New P2* as a part of the *comm state.* Then, $m3$ would be inserted to the front of the received-message-list of *New P2.* After the coordination, the communication connection between $P1$ and $P2$ is closed. $m6$ is considered as being sent from an unconnected process and would be appended to the end of the received-message-list of *New P2* accordingly.

4 Summary

We have introduced the design of data communication and process migration protocols for distributed computation. The protocols are built on top of a connection-oriented communication model where a connection is established prior to message passing. We have discussed mechanisms to handle various communication situations which can occur during process migration. First, we present schemes to capture messages in transit. The message coordination technique is employed to collect transmitting messages into receive-message-lists. Second, we discuss techniques to handle connection establishment during process migration. The sender-initiate technique where senders request for connections is employed. In case receivers migrate, the sender processes have to collaborate with the scheduler to find out the receivers' current locations. Finally, we discuss mechanisms to restore communication state so that message ordering is preserved. These mechanisms have been implemented in Snow's data communication and process migration protocols. Due to space limitation, full reports including implementation and experiments can be found in [5]. Analytical and experimental results confirm our design is valid and has a true potential in practice.

References

[1] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," tech. rep., Hewlett-Packard, Dec. 1998.

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[3] X.-H. Sun, V. K. Niak, and K. Chanchio, "A Coordinated Approach for Process Migration in Heterogeneous Environments," in *1999 SIAM Parallel Processing Conference*, Mar. 1999.

[4] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Mpvm: A migratable transparent version of PVM," *Computing Systems*, vol. 8, no. 2, pp. 171–216, 1995.

[5] K. Chanchio and X. H. Sun, "Communication state transfer for mobility of concurrent heterogeneous computing." Submitted for publication, 2001.

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed system," *ACM Transactions on Computer Systems*, pp. 63 – 75, 1987.