DESIGN, IMPLEMENTATION, AND FORMAL VERIFICATION OF ON-DEMAND CONNECTION ESTABLISHMENT SCHEME FOR TCP MODULE OF MPICH2 LIBRARY

A Thesis

by

SANKARA SUBBIAH MUTHUKRISHNAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Computer Science

DESIGN, IMPLEMENTATION, AND FORMAL VERIFICATION OF ON-DEMAND CONNECTION ESTABLISHMENT SCHEME FOR TCP MODULE OF MPICH2 LIBRARY

A Thesis

by

SANKARA SUBBIAH MUTHUKRISHNAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Computer Science

ABSTRACT

Design, Implementation, and Formal Verification of
On-demand Connection Establishment Scheme for TCP Module of MPICH2
Library. (August 2012)

Sankara Subbiah Muthukrishnan, B.E., Anna University, Chennai, India
Chair of Advisory Committee: Dr. Jaakko Järvi

Message Passing Interface (MPI) is a standard library interface for writing parallel programs. The MPI specification is broadly used for solving engineering and scientific problems on parallel computers, and MPICH2 is a popular MPI implementation developed at Argonne National Laboratory. The scalability of MPI implementations is very important for building high performance parallel computing applications. The initial TCP (Transmission Control Protocol) network module developed for Nemesis communication sub-system in the MPICH2 library, however, was not scalable in how it established connections: pairwise connections between all of an application's processes were established during the initialization of the application (the library call to `MPI_Init`), regardless of whether the connections were eventually needed or not.

In this work, we have developed a new TCP network module for Nemesis that establishes connections on-demand. The on-demand connection establishment scheme is designed to improve the scalability of the TCP network module in MPICH2 library, aiming to reduce the initialization time and the use of operating system resources of MPI applications. Our performance benchmark results show that `MPI_Init` in the on-demand connection establishment scheme becomes a fast constant time operation, and the additional cost of establishing connections later is negligible.

The on-demand connection establishment between two processes, especially when two processes attempt to connect to each other simultaneously, is a complex task due to race-conditions and thus prone to hard-to-reproduce defects. To assure ourselves of the correctness of the TCP network module, we modeled its design using the SPIN model checker, and verified safety and liveness properties stated as Linear Temporal Logic claims.

To my parents (Sornam and Muthukrishnan), wife (Suganya), and son (Sridarsh)

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisor, Dr. Jaakko Järvi, for his invaluable guidance, abundant patience, and consistent motivation throughout the research and writing. I am thankful to him for finding several gaps in the research and helping me address those. I would like to especially acknowledge his guidance on formal verification and benchmarks. Dr. Järvi has always been helpful in answering my questions by email even in the middle of the night. I am grateful to him for his tremendous help in reviewing the draft; without his conscientious feedback on structure, grammar, clarity, this thesis would not have been complete. I am thankful to him for his help in improving my technical writing skills. My special thanks to him for all his help with the logistics for the defense and approval process at A&M while I was working remotely.

I am indebted to Dr. Darius Buntinas for all his immeasurable help and guidance in the research from my internship days at Argonne National Laboratory through the consummation of the thesis. This work would not have been possible without his help and comprehensive and meticulous feedback on the design, implementation, tests, benchmarks, and formal verification throughout the research. I express my special thanks to him for helping to run several benchmarks on a HPC cluster at Argonne, and providing exceptional feedback on the write-up on several revisions without any hesitation. I am extremely pleased with the amount of assistance he extended to me after the internship. He is an excellent mentor for interns.

I would like to express my sincere thanks to my other committee members, Dr. A. L. Narasimha Reddy and Dr. Valerie E. Taylor for their time, valuable inputs, and co-operation. I extend my sincere thanks to Dr. William Gropp and Dr. Rajeev Thakur for their help in collaboration during my research at Argonne. I express my sincere gratitude to Dr. Gerard Holzmann for his invaluable time and help in responding to my questions, quickly, on using SPIN, iSPIN, and model checking, in general.

I gratefully acknowledge the computing resources provided on “Fusion,” a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

Friends and family play a key role. I would like to extend my gratitude to Sriram Ramanathan for reviewing a couple of chapters in the draft and his persistent motivation. I am grateful to Jayesh Krishna for his inputs and time for brainstorming on MPI , suggestions for the thesis outline, and constant encouragement. I sincerely appreciate Xiaolong Tang, ”lifesaver”, for his help in running around for signatures in several forms to meet every single deadline and thus help me graduate.

I would like to express my sincere gratitude to my managers at National Instruments, Mr. James R. Andrews and Mr. Michael Phillips, for allowing me to maintain flexible work schedule for a few weeks towards the completion.

I express my sincere gratitude to my mom Mrs. Sornam Muthukrishnan and dad Mr. Muthukrishnan Sudalai, for their love, sacrifice, motivation, and support to pursue an advanced degree. This thesis also would not also have been possible without my wife Suganya’s sacrifice, encouragement, and love. I particularly thank her for listening to my mock presentation and reviewing portions of the draft.

I would like to thank Rajkumar Kettimuthu for getting me an internship opportunity at Argonne and all his help. I express my sincere thanks to my friends Murali Karuppanan, Anuradha Mani, Balamurugan Muthuksamy, Shanmugavel Ponnusamy, Monish Mamadapur, Udayasarathy A Vijayasarathy, my brother Sundaramoorthy Muthukrishnan, and my sister Uma Rani Muthukrishnan for their endless encouragement and assistance in my pursuit of knowledge.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Parallel computers can deliver a large amount of computing power required by computationally intensive scientific applications such as modeling and simulations. In the past decade, requirements for computing power in many applications including commercial computing (online transaction processing) and entertainment industry (making computer-animated motion pictures) have grown notably and can only be delivered by high performance parallel and cluster computers. Parallel computing continues to become more and more ubiquitous and widespread. Due to several restrictions, including heat dissipation and power consumption, microprocessor speed and performance will likely not improve drastically in the near future. Major processor manufacturers have moved to hyper-threaded and multi-core architectures. Sutter [1] points out that concurrent programming is the next major revolution on how we write software.

There are several parallel architectures and a variety of hardware, such as SMPs, NUMA machines [2], massively parallel processors such as IBM Bluegene [3], and clusters [4]. In order to have efficient communication between the nodes in cluster systems, there are many network interconnects available which are based on different architectures with their accompanying protocols, such as Virtual Interface Architecture [5], Infiniband [6] and Quadrics [7]. A parallel application can achieve the best performance on a particular hardware architecture only by exploiting the specific features of that architecture. However, parallel applications should be written in such a way that they are portable across several architectures. Efficiently programming for

The journal model is *IEEE Transactions on Automatic Control.*

different parallel architectures in a portable manner is a considerable challenge. The Message Passing Interface (MPI) standard [8] is defined to address these portability and performance issues.

The MPI Standard is a message passing library standard, based on the consensus of the MPI Forum [9]. The MPI standard has become a *de facto* standard for writing parallel applications. There are several implementations available based on the standard; these include MPICH2 [10], OpenMPI [11], and LAM/MPI [12]. MPICH2 is one of the popular implementations, developed at Argonne National Laboratory, that supports several interconnects and architectures and provides an application programming interface (API) for the C, C++ and FORTRAN languages.

Performance and scalability are the chief design goals of the MPI libraries. As the size of the cluster computers grows, the scalability of parallel applications becomes more and more important. This thesis focuses on the impact that initiating connections between different processes in an MPI application can have on scalability.

Two processes in a parallel application pass messages between each other by establishing a connection between them. Even though the MPI libraries are efficient and scalable in general, some of them are not scalable in how they establish connections. These MPI implementations create connections between all the processes during initialization of the parallel application (i.e., during the MPI library call `MPI_Init`). However, many parallel applications do not require connections between all the processes in the application. Each process in a parallel application may thus consume system resources for connections that are never used and cause depletion of system resources. Moreover, initialization may require a considerable amount of time, which is also not desirable in some applications.

Furthermore, some applications have further demands, such as checkpointing [13, 14], a relatively new feature being introduced in the MPI libraries. Checkpointing allows suspending a parallel job either in order to schedule a higher priority job on a supercomputer or in the event of a hardware or software failure, and resuming the job from the point of suspension without losing any of the previous work. Parallel jobs that are checkpointed need to tear down all the connections including the never-used connections, and need to initialize all the connections while restarting the checkpointed job, and both can be very time-consuming.

All of the aforementioned issues diminish the scalability of the parallel applications. These issues are alleviated if the connections are established between processes only when needed, rather than during initialization time. In such an *on-demand connection establishment scheme*, the creation of a connection is delayed until a process tries to send a message to or receive a message from another process. In this work, we have developed such a scheme for the TCP network module of the Nemesis channel, which is a communication sub-system designed for both high-performance intra-node (using shared memory) and high-performance inter-node communication (using networks such as TCP) integrated into MPICH2 library.

We originally designed and implemented on-demand connection establishment scheme as a stand-alone prototype. During the design phase of the prototype, we underestimated the complexity, which became apparent when implementing the prototype, especially during its integration into the TCP network module of the Nemesis channel. Also, the researchers at Argonne National Laboratory had past bitter experiences — while maintaining MPICH2 library software — with several hard-to-track race-condition related bugs in the state machines in the old communication channels that existed before the Nemesis channel. In order to be confident about the correctness of the on-demand connection establishment scheme and the state machine and to ensure the reliability and robustness of the new TCP network module and thus ease its maintenance, we decided to rely on formal verification techniques, *model checking* in particular. The state-of-the-art model checker SPIN [15] was chosen for this task. The state machine describing the behavior of our on-demand connection establishment scheme was modeled in the PROMELA language. Several safety properties of this model were formulated as Linear Temporal Logic claims [16,17] and verified using the SPIN model checker. The PROMELA model was also verified using the built-in safety verification algorithm of SPIN to ensure that there are no deadlocks in the model.

The contributions of this thesis are:

- The design of the on-demand connection establishment scheme for the Nemesis channel in TCP module of MPICH2.
- A prototype implementation, tested for correctness and scalability, of the state machine that realizes the on-demand connection establishment scheme.
- The integration of the state machine implementation into MPICH2's TCP network module.
- The development of benchmarks that compare the on-demand and static connection schemes.
- A model of the state machine in PROMELA language.
- The specification and verification of several safety and liveness properties using the SPIN model checker.
- Checking for redundancies in the model using SPIN's *property-based slicing techniques* [18].

The thesis is structured as follows: Chapter II describes the relevant background information about programming with TCP sockets and MPI. Chapter III elucidates the scalability problem of MPI applications and briefly outlines how this problem can be solved by establishing connections on-demand. Chapter IV explains the design and implementation of the state machine in connection establishment. In Chapter V, we review the relevant background to the model checker SPIN, its language PROMELA, construction of the PROMELA model of the state machine, and verification of safety and liveness properties using Linear Temporal Logic claims. In Chapter VI, we discuss the benchmark results that show how `MPI_Init` time is reduced with the on-demand connection establishment scheme. Chapter VII concludes this thesis and outlines some potential future work, especially related to model checking.

CHAPTER II

BACKGROUND

This chapter discusses some relevant background material for our work in this chapter. We first briefly review TCP/IP sockets and socket programming with the focus on connection-oriented sockets. We then present an overview of MPI and the MPICH2 library.

A. Sockets

The TCP/IP network stack implementation exposes its services to applications through the socket programming interface, which has evolved to be the *de-facto* standard of network programming. The socket programming interface is a part of the IEEE POSIX standard [19] that enables the implementation of portable network applications across several platforms. We briefly review sockets in this section; a good source for further information is the book "Unix Network Programming — Volume 1" [20].

Most of the network applications use one (or both) of the two types of sockets:

– **Datagram sockets**: UDP (Universal Datagram Protocol) is a connectionless transport layer protocol in the network stack. This protocol provides the concept of *application endpoint* (UDP port) to its clients. The endpoint allows applications to identify the remote destination to which data needs to be sent or from which received. Since UDP is a connectionless protocol, it does not incur a handshake overhead for setup or tear-down. However, UDP does not support reliability(guaranteed delivery of packets without errors and duplicates), sequencing of packets, and retransmission of packets in case of packet loss or corruption. UDP is useful in many applications where speed is more important than reliability and receipt of packets out-of-sequence is not a problem. Higher-layer network protocols (like DHCP and DNS) are implemented using the UDP protocol services. In the socket parlance, a socket that uses UDP is commonly known as a *datagram* socket.

– **Stream sockets**: TCP (Transmission Control Protocol) provides guaranteed in-order (sequenced) data delivery service to its clients. Like UDP, TCP also provides an application endpoint (TCP port) to its clients. However, unlike UDP, TCP is connection-oriented, which means that two applications must establish a connection explicitly before they can exchange data. TCP connections are full-duplex, with each endpoint having its own data stream to the other endpoint. TCP also provides congestion control [20] and flow control [20] between the communicating endpoints. TCP sockets are also called stream sockets. We discuss the TCP sockets briefly in the next section.

1. TCP Socket Programming

TCP socket programming typically follows the client server programming model. A socket can be thought of as a communication endpoint. A socket endpoint is uniquely identified by the combination of an IP address and a TCP port. A socket pair (client endpoint and server endpoint) forms an active TCP socket connection. The server listens on a port; the client needs the port number to request a connection; for standard services such as http, ssh, etc, the ports are standardized/well-known; for other services, the port needs to be communicated to the client in some way. The client connects to the server at the server's advertised port, and then the server and the client start communicating with each other. A system, or, to be precise, a process running in a system, can act both as a server and a client.

We introduce some of the key socket functions with an example consisting of a simple client and a server. The client sends a text string to the server; the server echoes back any message that it receives. The client program is outlined in Listing II.1 and the server program in Listing II.2. These programs are not complete; some source code is omitted and error checking of functions is not done for brevity.

Listing II.1 TCP client program.

```
int main(int argc, char *argv[])
{
  int                conn_sock; //  connection socket
  struct sockaddr_in servaddr;  //  socket address structure

  //  Create the stream socket
  conn_sock = socket(AF_INET, SOCK_STREAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family      = AF_INET;
  servaddr.sin_port        = htons(10000);
  servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");

  //  connect to the remote echo server
  connect(conn_sock, (struct sockaddr *) &servaddr,
    sizeof(servaddr));

  while(1) {
  // Send a string to the server.
  // Receive the string echoed by the server.
  // source code omitted for brevity
  }
  close(conn_sock);
}
```

Listing II.2 TCP server program.

```
int main(int argc, char *argv[])
{
  int                lstn_sock; // listening socket
  int                acpt_sock; // accepted connection socket
  struct sockaddr_in servaddr;  // socket address structure

  //Create a listening socket.
  lstn_sock = socket(AF_INET, SOCK_STREAM, 0);

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(10001);

  //Explicitly bind to port 10001.
  bind(lstn_sock, (struct sockaddr *) &servaddr,
    sizeof(servaddr));

  //Listen for incoming connections.
  //Make the backlog as much as the stack will support.
  listen(lstn_sock, SOMAXCONN);

  while (1) {
    acpt_sock = accept(lstn_sock, NULL, NULL));
    // Receive a string from the client.
    // Send it back to the client.
    // source code omitted for brevity.
    close(acpt_sock);
  }
}
```

We start with the client as it is the simpler of the two programs. The client first creates a socket using the `socket` function. When a socket is created, it is not assigned an endpoint. An endpoint can be assigned to a socket in two ways: explicitly using the `bind` function, or implicitly using the `connect` function. Since the client is not concerned which port in the local machine it uses, it relies on the `connect` function to assign it an endpoint. The `connect` function establishes a connection to the server and assigns an endpoint to the client socket. We refer to this socket as the *connecting socket*. After the socket gets connected to the remote server, the client sends the request message to the server using the `send` function and, then, receives the message from the server using the `recv` function.

The TCP server begins by creating a socket. It uses the `bind` function to attach an endpoint (in our case, the port of 10001 and IP address of the local host). It then calls the `listen` function that makes the socket a passive socket, i.e., a socket that can be used to only accept new connection requests. Subsequently, the server calls the `accept` function on the listening socket that waits for a new connection request from the client, and returns a new socket when such a connection request is found. This new socket is the one that can be used to communicate with the client; we refer to this socket as *accepting socket*. The server receives the message from the client using the `recv` function and sends it back to the client using the `send` function. Figure 1 shows the end-points of *connecting socket* in the client process and *accepting socket* in the server process and how full-duplex communication happens between the two processes.

2. Issues with Blocking Sockets

In the simple client-server example we discussed above, the `connect`, `accept`, `send`, and `recv` functions are blocking calls. In other words, they will block in the operating system's network stack until the function succeeds. Thus, this model can have both reliability and performance issues. For instance, the server hanging after accepting the connection from the client can cause the client to block indefinitely until the server is shut down. The server or the client process cannot do any other useful work when they are blocked in one of the functions.

There are two solutions for the aforementioned issues. One solution, for instance for the server side, is to spawn a new thread to handle each client. If one client hangs, this does not prevent the server from serving other clients. The drawback with this approach is that threads can be expensive resources in some systems. This approach can also complicate the application design. The other solution is to use non-blocking sockets. We will review non-blocking sockets in the next section.

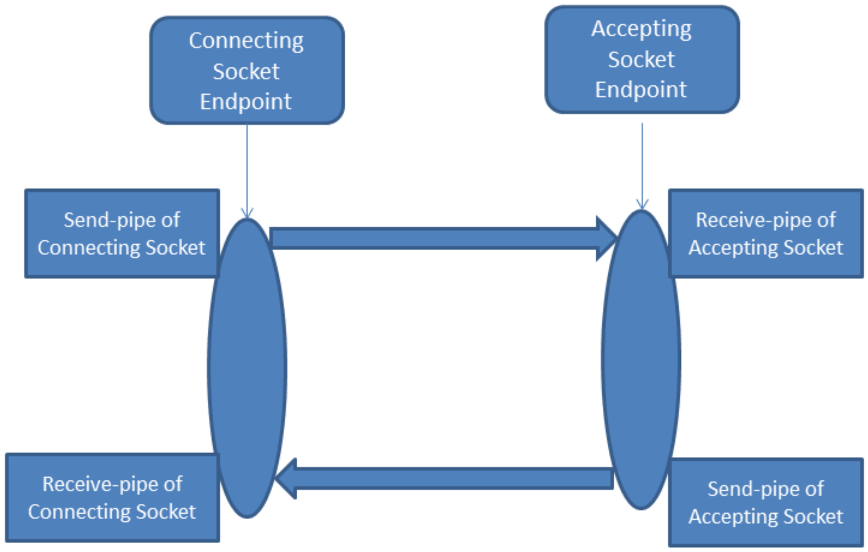
Connecting Socket Endpoint
Accepting Socket Endpoint
Send-pipe of Connecting Socket
Receive-pipe of Accepting Socket
Receive-pipe of Connecting Socket
Send-pipe of Accepting Socket

Fig. 1. Connecting socket and accepting socket end-points.

3. Non-blocking Sockets

After a socket gets created using the `socket` function or the `accept` function, it can be made non-blocking using the function `fcntl`. After a socket is made non-blocking, the socket functions such as `connect`, `accept`, `send`, and `recv` will return immediately with a special return code (such as EAGAIN or EWOULDBLOCK) if they cannot succeed, and the application must either retry later, or check whether the pending operation has succeeded later. The application, instead of simply retrying, can call the `poll` function that indicates whether a socket is ready for an I/O operation (sending, receiving, or both) or whether there are errors in the socket.

We briefly discuss the non-blocking functions. If a `connect` function is called on a socket and a connection cannot be established immediately, this function returns an error indicating that the connection is in progress. The application can call `poll` later to determine whether the connection has been established. Similarly, a listening socket can be checked using `poll` to see whether there is a new connection request pending before the `accept` function is called on a listening socket. A non-blocking application checks whether a socket is readable before trying to receive data from the socket, and checks whether a socket is writable before trying to send data.

Non-blocking sockets are inherently more complex to design and implement: buffer management gets tricky because read and write to sockets can return when they are partially complete, and socket implementations in various platforms handle errors differently for non-blocking functions, making portability an issue.

B. A Tutorial on MPI

Message Passing Interface (MPI) standard [8] has been evolving since 1992. It has matured significantly in functionality and it is widely supported on almost all of the

HPC (High Performance Computing) platforms. Several vendor implementations and open-source implementations [10–12] are available, and they exploit various processor and network interconnect features to optimize the performance. The programming model of MPI requires explicit parallelism; in other words, the developer of a parallel application is responsible for identifying parallelism and designing the application using MPI functions to exploit parallelism. Even though the MPI model was originally developed for the distributed memory architecture, the model naturally works for the shared memory architectures, such as SMP and NUMA [2].

1. An MPI Program

This section briefly reviews programming with MPI, starting with a walk-through of a simple MPI program in Listing II.3. The MPI program first calls the function `MPI_Init` before calling any other MPI functions. `MPI_Init` initializes the MPI execution environment in the context of all the processes in the application. After initialization, MPI processes typically compute and exchange data with each other. For simplicity, our sample program does not do any computation. In order for one process to identify another process to which data needs to be sent to or received from, each process is given a unique identifier during `MPI_Init`, known as the rank of the MPI process. `MPI_Comm_rank` function can be used to query the rank of the current process. A typical parallel application may have the need to form different small groups of processes, either to communicate with each other process within a group, or across groups. MPI provides an abstraction called *a communicator* for this purpose. There is one default communicator called `MPI_COMM_WORLD` that includes all the processes in the application. The size of a communicator can be obtained using `MPI_Comm_size`. In the example in Listing II.3, each MPI process other than the one with rank zero sends (using `MPI_Send`) one character in the send buffer (indexed by its rank) to the process with rank zero. The process with rank zero receives a character from all the other processes (using MPI_Recv) and puts that character into the receive buffer at the location indexed by the rank of the sending process. The MPI application calls MPI_Finalize which cleans up the resources in the execution environment of all the processes. As the name suggests, no MPI library function shall be called after MPI_Finalize.

Listing II.3 Hello World MPI program.

```
int main(int argc, char *argv[])
{
  int myRank, commSize, maxRank;
  char sbuf[] = "Hello World";
  char *rbuf;

  MPI_Init(0, 0);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &commSize);

  maxRank = MIN(strlen(sbuf), commSize);
  if (myRank == 0) {
    int remoteRank;
    MPI_Status status;

    rbuf = (char *) malloc(strlen(sbuf)+1);
    rbuf[0] = 'H';
    for (remoteRank = 1; remoteRank < maxRank; remoteRank++)
      MPI_Recv(&rbuf[remoteRank], 1, MPI_CHAR, remoteRank,
         100, MPI_COMM_WORLD, &status);
    rbuf[remoteRank] = '\0';
    printf("%s\n", rbuf);
    free(rbuf);
  }
  else if (myRank < maxRank) {
    MPI_Send(&sbuf[myRank], 1, MPI_CHAR, 0, 100,
       MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

2. Communication Functions

In the aforementioned example, we have used two communication functions called `MPI_Send` and `MPI_Recv`. The MPI communication functions can be broadly classified into two categories: point-to-point communication and collective communication. Point-to-point communication, as the name suggests, always happens between two MPI processes. This communication supports different flavors that can be classified orthogonally: (1) blocking and non-blocking, (2) synchronous, buffered, and ready. Collective communication must involve all the processes in a communicator, and all collective MPI functions are blocking. For a detailed overview on different types of communication and other advanced features, consult the books "Using MPI" [21] and "Using MPI-2" [22].

3. Communicators

A communicator is a group of processes that may communicate with each other. All MPI functions that exchange data must specify a communicator. Communicators allow the parallel application developer to organize tasks into different groups based on the computation functions that need to be performed by them. The communicator exposes the notion of user-defined *virtual topologies*, which is useful in the design of parallel applications. Since collective communications need to be performed by all the processes in a group, the concept of communicator lends itself nicely to collective communication. Communicators are dynamic, and they can be created and destroyed during the lifetime of an application; a process can be part of more than one communicator, however, it has a unique rank in each communicator. For details on using communicators, consult the MPI reference documentation [21, 22].

4. MPICH2 and Nemesis

MPICH2 library [10] is a popular implementation of MPI. It supports different clusters (shared-memory, multi-core), high-speed network interconnects (Gigabit ethernet, Infiniband [6], Myrinet [23] and Quadrics [7]), and proprietary high-end computing systems (Blue Gene, Cray). MPICH2 supports the features of both MPI standards, MPI-1 and MPI-2 [8]. A high-level architecture depicting different layers of MPICH2 is given in Figure 2. ADI-3 is a full-featured *abstract device interface* that provides a portability layer to allow access to many performance-oriented features of several communication systems [24]. *CH3* [24] is a simplified *channel device* that requires the implementation of only a dozen functions but provides many performance advantages of the ADI-3 interface. Modules such as "TCP/IP" and "shared memory" are implemented as CH3 *channels*.

Nemesis [25,26] is a more recent development of MPICH2. It is the communication sub-system, redesigned by Buntinas, Gropp, et al., to improve the performance and scalability of MPICH2; it uses shared memory for communication between processes in the same node (intra-node) and a network module such as TCP for communication between processes running in different nodes (inter-node). Though the intent is to ideally implement Nemesis as an ADI-3 device, Nemesis is integrated as a CH3 channel (for quicker prototyping and experimenting). The integration of Nemesis as a CH3 channel and Nemesis supporting different network modules such as TCP, GM [23], and ELAN [7] are shown in Figure 2.

5. Virtual Connection and Network Module Connection

Ranks are used to communicate between processes within the same communicator. If one process in one communicator needs to communicate with another process in a

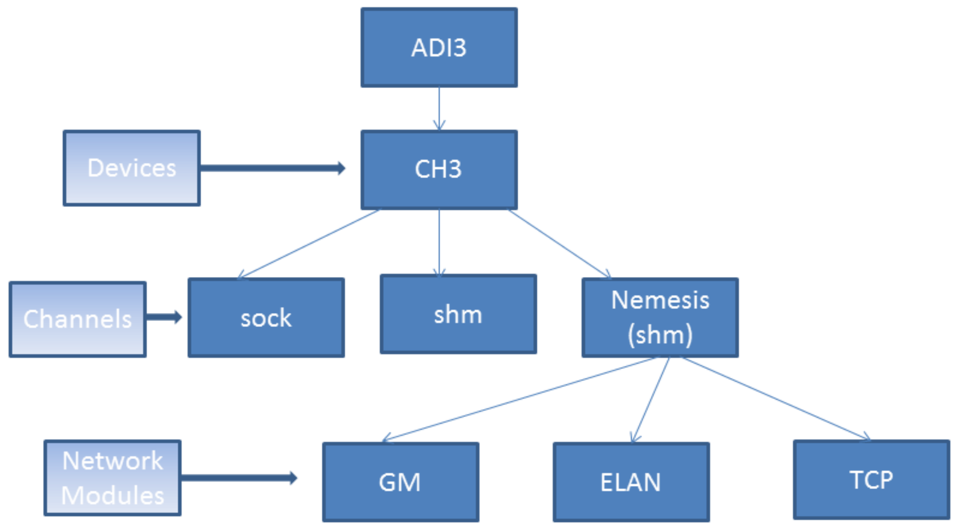

Fig. 2. Architecture of MPICH2.

different communicator, it uses both rank and *communicator identifier* of the remote process. The communicator identifier is referred to as *process group identifier* (or simply *PGID*) in the MPICH2 implementation. In MPICH2, the communication between processes using ranks and PGIDs is abstracted at the CH3 device level. CH3 device abstracts communication to a remote MPI process as a *virtual connection*. The network module such as TCP establishes the physical connection to the remote process using the appropriate network protocol. When the network connection is established, the virtual connection and the network connection (such as socket connection) are associated with each other. Figure 3 shows how a virtual connection and one of the network module connections is associated.

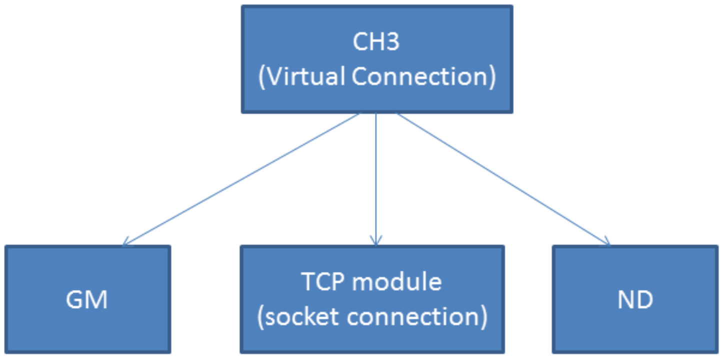

Fig. 3. Virtual connection and network module connection.

CHAPTER III

SCALABILITY ISSUES AND A SOLUTION

A. Scalability Problem

MPICH2 provides an efficient MPI-2 implementation for small- and large-scale clusters, SMP machines, and massively parallel processors such as IBM Bluegene [3]. While one of the goals of the MPI standard, apart from improving portability of parallel applications, is to bridge the gap between the marked performance offered by different parallel architectures and the actual performance delivered to the application, the extent to which this is achieved depends on the implementation. Our focus is on Nemesis [25, 26], a communication subsystem of MPICH2 library, designed and implemented by Buntinas, et al. after identifying the critical areas to improve the performance of MPICH2.

Nemesis was designed to be a scalable, high-performance, shared-memory, multi-network communication subsystem for MPICH2. The design goals, in order of priority, were scalability, high-performance intra-node communication, high-performance inter-node communication, and multi-network inter-node communication [25]. Nemesis uses shared-memory for intra-node communication and a network module such as TCP for inter-node communication. Nemesis was successfully integrated into MPICH2 and yields very good performance [25]. The main focus in the initial implementation of Nemesis was on efficient and high performance communication. Because of this focus, some other areas received less attention. In particular, the connection establishment was not scalable. In the original implementation of the network modules of Nemesis, all connections were *statically* created during `MPI_Init`. As a matter of fact, several MPI implementations statically create connections during `MPI_Init`.

This connection management scheme is called *static connection establishment scheme*, and it is also known as *eager connection establishment scheme*. It is easy to implement this scheme, but it suffers from scalability and performance issues.

We will review the several disadvantages of static connection establishment scheme below:

- If a parallel application has N processes and each process connects to all the other processes, the parallel application will have $\Theta(N^2)$ connections, $N \times (N-1)/2$ to be exact. Most large-scale parallel applications are not fully-connected. Table I (reproduced from [27]) lists the average number of destinations each process has in different parallel applications. The estimation of 1024 processes [27] is based on communication patterns described in [28] and implementation of collective communication using binomial tree algorithms [29]. The average number of connections each process makes in a 1024 process parallel application is typically less than 11 [28]. Therefore, in several parallel applications with the static connection establishment scheme, a large amount of resources gets allocated in creating $\Theta(N^2)$ connections which are never used.

- Establishing $\Theta(N^2)$ connections during initialization of a parallel application, in other words, during the library call `MPI_Init`, consumes a lot of CPU cycles and the parallel application will have a very slow start. While the start-up time for some of the applications may not be crucial, it is very important for certain class of applications.

- The number of established connections affect not only the start-up time of an application, but also its shutdown (either normal of forceful) time. The more connections that have been established, the more connections there are that need to be torn down. This has significant impact for parallel applications that

Table I. Average number of distinct destinations per process.

Application	Number of Processes	Average Number of Destinations
sPPM	64	5.5
	1024	< 6
SMG2000	64	41.88
	1024	< 1023
Sphot	64	0.98
	1024	<= 1
Sweep3D	64	3.5
	1024	<= 4
Samari4	64	4.94
	1024	<= 10
CG	64	6.36
	1024	<= 11

are checkpointed. When a higher priority (even a mission-critical) parallel job should be scheduled on a cluster/supercomputer after checkpointing the currently running parallel application, it is imperative that the current application is checkpointed and shut down gracefully as quickly as possible. In addition to that, a checkpointed application should also be resumed fast when it is rescheduled by the job scheduler. When the number of connections, however, is large, checkpointing and restarting a parallel application takes a significant time.

- Unnecessary use of system resources in a fully-connected application can also cause performance degradation during the execution of the application. Typically the MPI library maintains a table of connections internally. If the number of connections is large, as is the case in the static connection establishment scheme, the number of connection entries in this table may take up considerable space, and accessing connection entries can lead to page-faults. If a process's table of connections consist of only those that are used, the table will be smaller and may fit into fewer memory pages; typically even into one page.

B. Solution

We have discussed several problems in establishing connections statically during the initialization of a parallel application. This thesis demonstrates that by establishing connections as needed, the problems can be solved. Concretely, during `MPI_Init`, no connections are created between any processes. A connection is only created between two processes when they try to communicate with each other using MPI functions such as `MPI_Send` and `MPI_Recv`. This scheme is called the *on-demand connection establishment scheme* or, alternatively, the *lazy connection establishment scheme*. We briefly outline this scheme and discuss some of its intricacies in this section. The discussion assumes knowledge of network programming with sockets, socket functions and several MPI concepts, including virtual connections, and the identification information of MPI processes. Please consult Chapter II for a brief review of these topics.

Each MPI process has a *listener* socket that accepts connections from other processes. When an MPI process `P1` tries to establish a virtual connection to another MPI process `P2` in order to send or receive data, `P1` creates a network socket and issues a `connect` to `P2`. The listener socket in `P2` that issues an `accept` call establishes the connection with `P1`. Then, `P1` sends its identification information to `P2` that helps `P2` to associate the newly created socket connection to a virtual connection. As `P1` initiates the socket connection for its virtual connection, it trivially associates its virtual connection with the established socket connection. Similarly, when `P1` wants to disconnect a virtual connection with `P2`, it does so by closing the associated socket connection. At the first glance, this looks trivial. However, this scheme gets complex when two communicating processes race with each other to connect and disconnect sockets. We will briefly discuss the intricacies of this scheme in the next subsection.

1. Head-to-head Resolution of Connections

We explained in Chapter II that a socket connection between two processes is duplex, and therefore each process can send and receive data from another process simultaneously using only one socket. Hence, one virtual connection of a process should be associated with one socket connection. It will be redundant to keep two socket connections between a pair of communicating processes; also, the design and implementation to associate two socket connections (for that matter, two of any network module connections) to a virtual connection will add unnecessary complexity in the design and implementation. We will show a scenario that results in two socket connections when two MPI processes try to connect to each other at the same time.

In Figure 4, two processes `P1` and `P2` try to connect to each other at the same time, send their identification information to each other, and create two network connections (one that is initiated and the other that is accepted) that can be used by the virtual connection. This problem may arise in a naïve implementation of supporting on-demand connections. One simple solution is to check whether one connection is a duplicate and close the duplicate connection. An instinctive attempt to solve this problem is shown in Figure 5, and discussed below.

In Figure 5, two processes `P1` and `P2` connect to each other simultaneously and both of them create two network connections as explained above, and illustrated in Figure 4. Later, `P1` realizes that it has a duplicate connection and closes the one it accepted from `P2`. `P2`, while noticing the duplicate connection, closes the one it accepted from `P1`. Both the processes close both of their connections, which results in not having any network connection at all between the two processes. We refer to these kinds of issues as head-to-head connection establishment issues. The above two scenarios are just examples we have used for illustration, and by no means exhaustively cover all the head-to-head situations.

The head-to-head connection issues do not arise in the static connection establishment scheme; each process initiates a connection only with the higher-ranked processes, thus avoiding duplicate connections. We find that resolving head-to-head connections is not trivial but requires careful design. In the next chapter, we will discuss the detailed design and implementation of the on-demand connection establishment scheme.

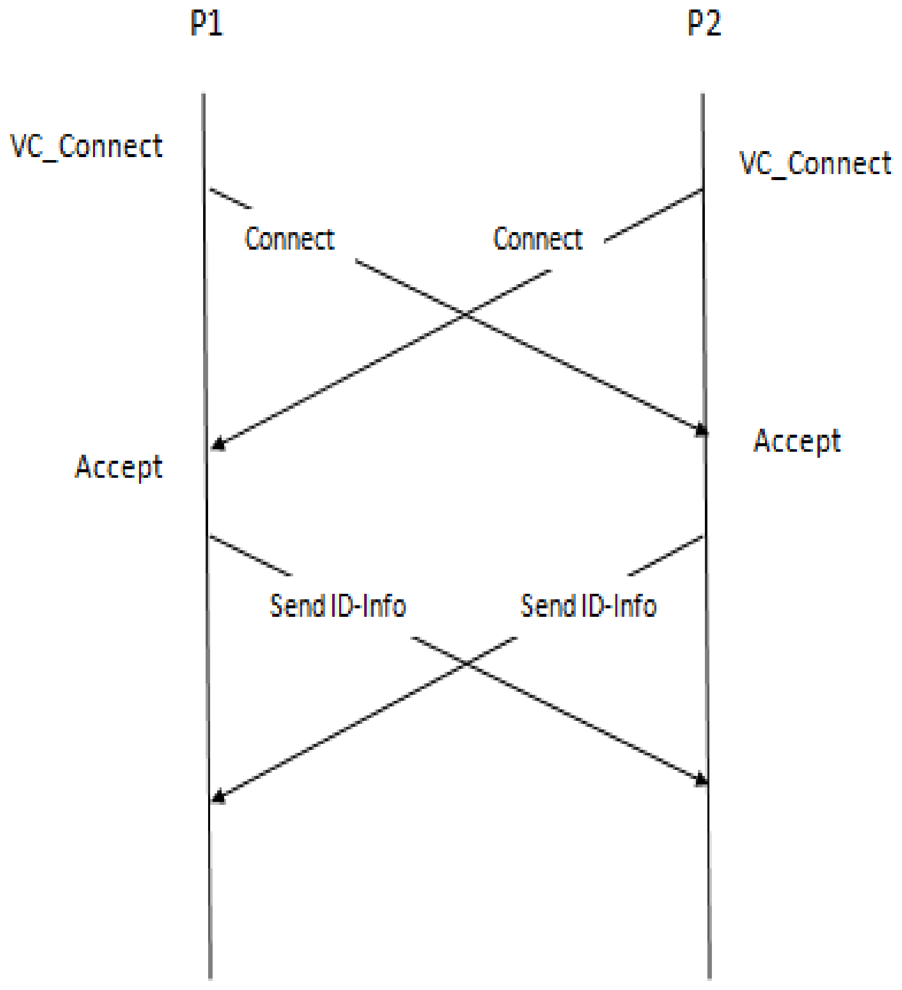

Fig. 4. Head-to-head connections resulting in duplicate connections.

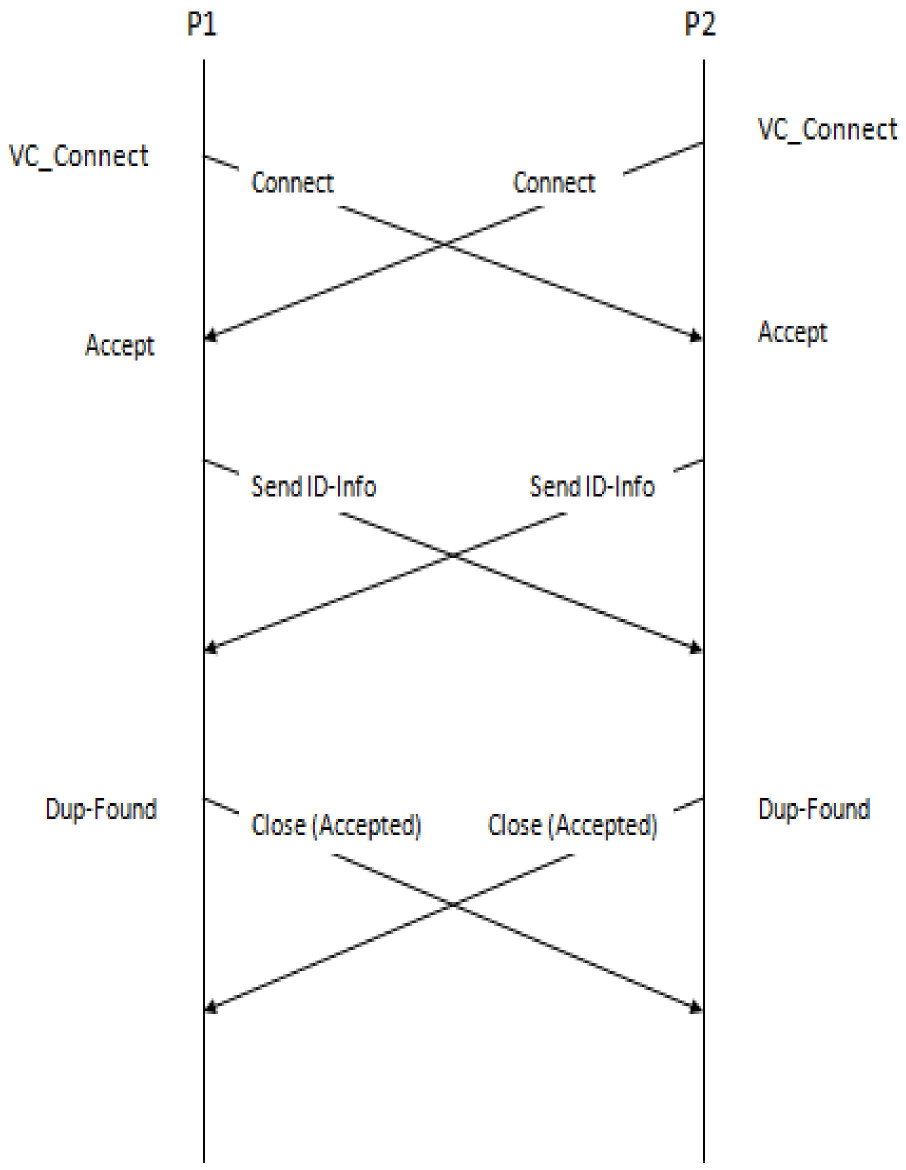

Fig. 5. Head-to-head connections resulting in no connections.

CHAPTER IV

DESIGN AND IMPLEMENTATION

We discussed how head-to-head resolution of connections is important and not trivial in Chapter III. It was important for us to design a state machine and handle various states of an MPI process to resolve the head-to-head situations. In this chapter, we will discuss the design of the on-demand connection establishment scheme and some implementation details.

In our work, we have chosen TCP sockets as the services TCP sockets provide (as discussed in Chapter II) are important to the TCP network module. One of the key decisions before designing a state machine is to choose between blocking and non-blocking sockets. MPICH2 library implementation handles both communication requests and connection requests without using any additional dedicated threads. This is possible with an approach that is based on polling; by periodically checking whether communication and connection requests are pending and making progress on them. Therefore, we have chosen non-blocking sockets for the TCP network module. The alternative design option — adding new threads to handle the connection requests — would require significant amount of redesign of MPICH2, and therefore is not chosen.

Each MPI process has to accept connections from other MPI processes. Hence, when the TCP network module is initialized, we create a listener socket that listens for new connections in a polling loop. We keep a table of all the socket connections that are both initiated by the process and that are accepted by the process. Each entry in this table has the following fields: a socket descriptor, a boolean that indicates whether the peer of the connection is in the same process group, the rank of the peer in the process group, the process group ID of the peer if the peer is not in the same process group, the current state of the connection, the handler function for the current state, and a pointer back to the virtual connection. Some fields, such as the rank and the process group ID, are cached in this structure even though they can be obtained from the pointer to the virtual connection; this avoids the performance penalty of chasing pointers. We also keep another table of *struct pollfd* structures [20] with all the socket descriptors for polling using the system call `poll`. Each entry in these two tables refers to the same socket connection.

The state machine has several states (to be explained in detail later in this section) and there is a handler function for each state. The progress engine polls for all the socket connections in the polling table, and, if an event has occurred for a socket descriptor, it calls the handler function of that socket connection to take appropriate action (including progressing to another state in the state machine for most of the states). We have discussed in Chapter II that when a socket is polled using the `poll` system call, we can query specifically whether the socket is ready for *reading* or *writing* or both. We have optimized the polling performance by requesting specific events expediently in the call to the `poll` system call based on the current state of the state machine.

As the state machine is too large to fit into one page of this report, they are split into two state machines, one for the *connect-side* and another for the *accept-side*. The connect-side of the state machine is depicted in Figure 6 and the accept-side in Figure 7. For clarity and a holistic picture, we show the common states of both machines in Figure 8. As the state machine is non-trivial and split into two figures, the reader may have to refer to all the three figures and read the explanation of the states and handler functions back and forth. To help in this task, we use the following conventions in prefixing the names of the states:

`TS_` : a shared state of both the connect-side and the accept-side state machines

`TC_` : a state of the connect-side state machine

`TA_` : a state of the accept-side state machine

`_C_` : a state that is part of connection sequence

`_D_` : a state that is part of disconnection sequence

In the state machine diagrams shown in Figures 6, 7, and 8, each transition is labeled with "event/action". Each transition may be read as the following: If "event" occurs in the current state, perform the "action" and transition to the new state.

The following is a list of all the states, each accompanied with a short description.

– `TS_CLOSED`: This is the initial state of a socket connection for both the connect-side and the accept-side state machines. The network socket does not simply exist in this state.

– `TC_C_CNTING`: After a non-blocking `connect` is issued on a socket, if the socket does not get connected immediately to the peer, then the state-machine transitions to this state.

– `TC_C_CNTD`: After `connect` is issued on a socket and the socket gets connected to the peer, the state machine goes to this state.

– `TC_C_RANKSENT`: When the identification information of the MPI process is sent to the peer from the `TC_C_CNTD` state, state machine transitions to this state.

– `TA_C_CNTD`: When there is a new connection in the listen queue and it is accepted, the state machine of the connection moves to this state.

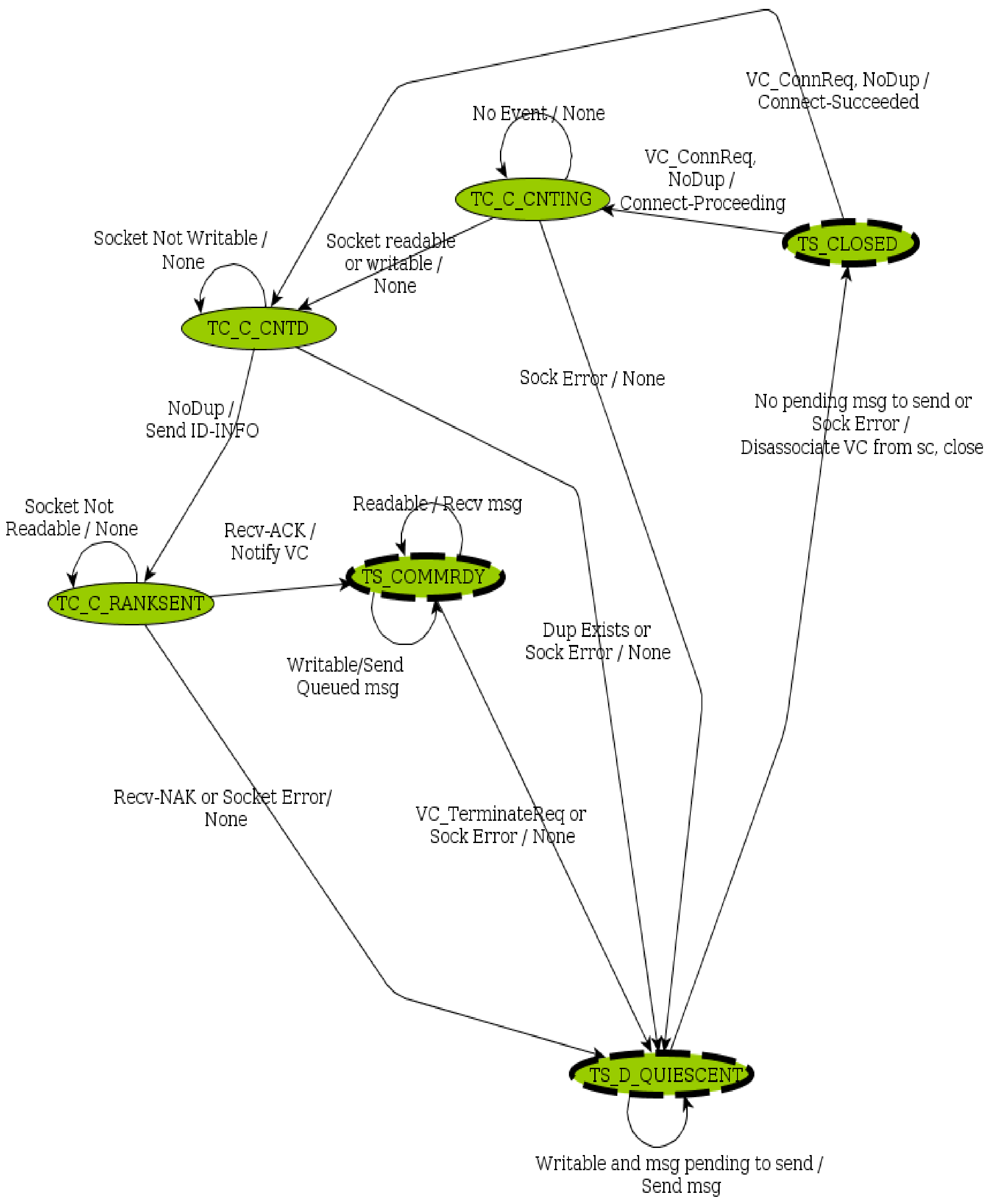

Fig. 6. Connect-side state machine of the TCP network module.

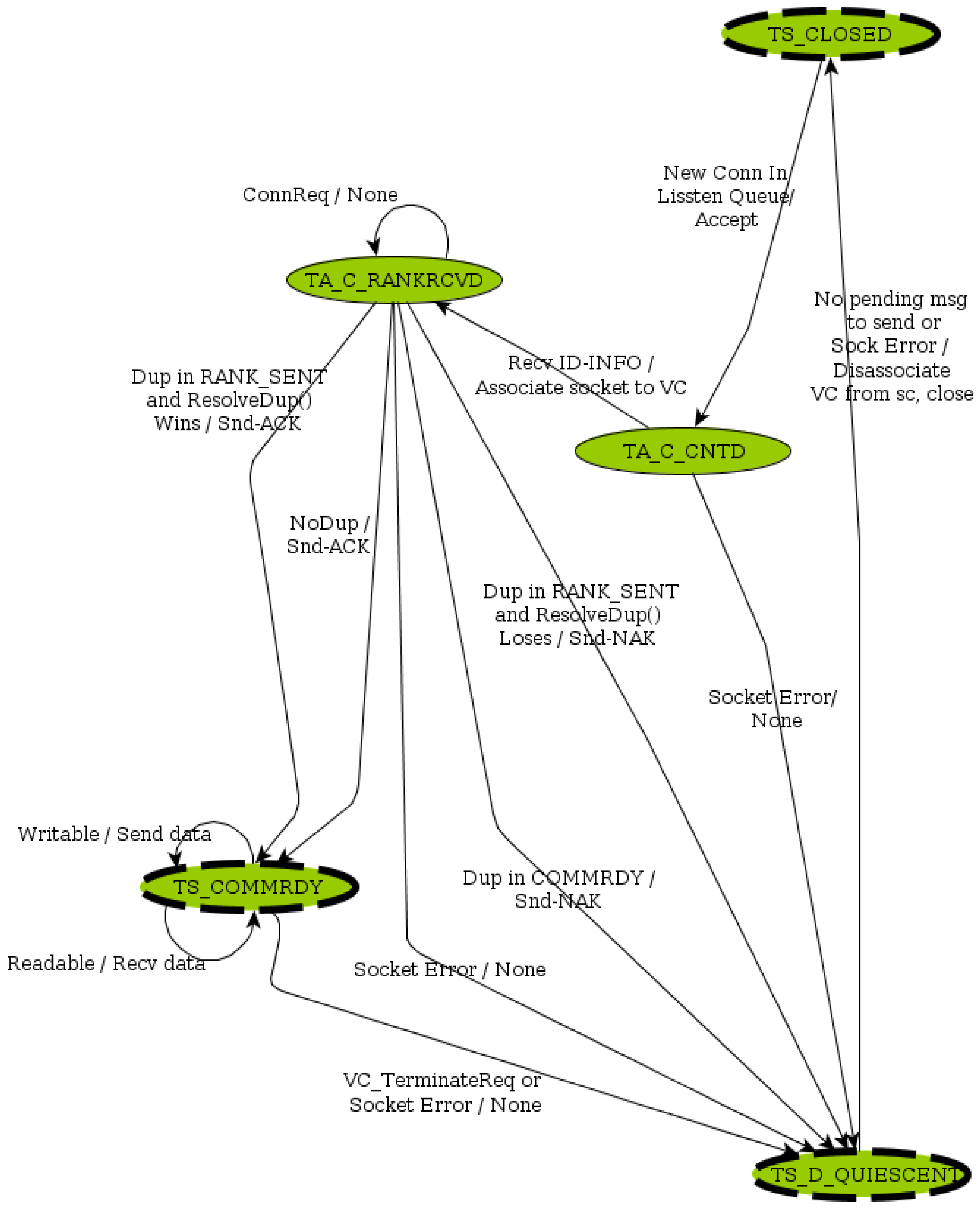
TS_CLOSED
New Conn In Lissten Queue/ Accept
No pending msg to send or Sock Error / Disassociate VC from sc, close
ConnReq / None
TA_C_RANKRCVD
Recv ID-INFO / Associate socket to VC
TA_C_CNTD
Dup in RANK_SENT and ResolveDup() Wins / Snd-ACK
NoDup / Snd-ACK
Dup in RANK_SENT and ResolveDup() Loses / Snd-NAK
Socket Error/ None
Writable / Send data
TS_COMMRDY
Readable / Recv data
Dup in COMMRDY / Snd-NAK
Socket Error / None
VC_TerminateReq or Socket Error / None
TS_D_QUIESCENT

Fig. 7. Accept-side state machine of the TCP network module.

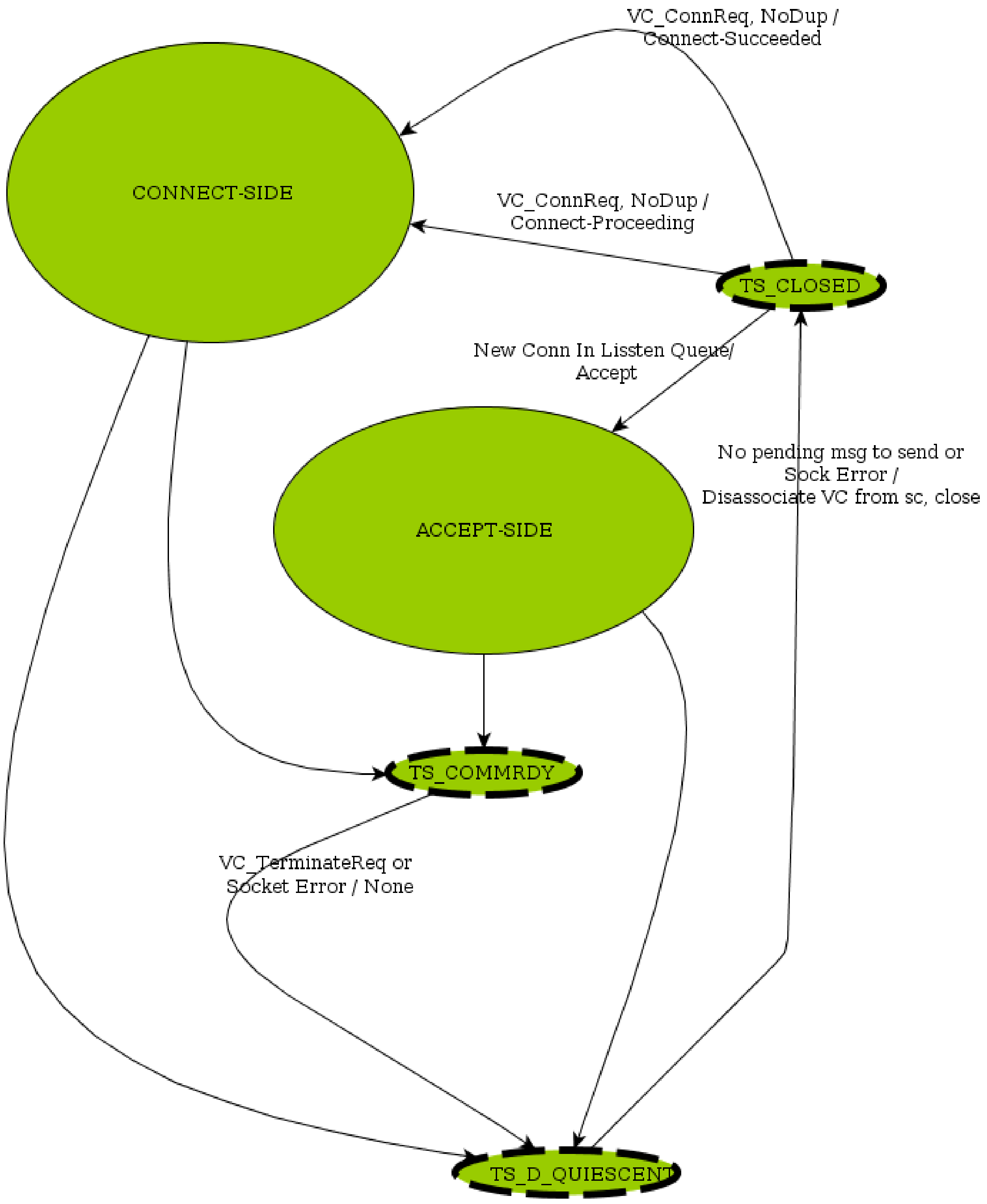
VC_ConnReq, NoDup / Connect-Succeeded
CONNECT-SIDE
VC_ConnReq, NoDup / Connect-Proceeding
TS_CLOSED
New Conn In Lisstten Queue/ Accept
No pending msg to send or Sock Error / Disassociate VC from sc, close
ACCEPT-SIDE
TS_COMMRDY
VC_TerminateReq or Socket Error / None
TS_D_QUIESCENT

Fig. 8. Common states of connect-side and accept-side state machines.

- `TA_C_RANKRCVD`: When the identification information is received from the peer in the `TA_C_CNTD` state, the state machine transitions to this state.

- `TS_COMMRDY`: After the state machine transitions to this state, the communication may happen with the peer process. The associated virtual connection of this socket connection is notified that sends and receives may happen now on this socket connection.

- `TS_D_QUIESCENT`: The state machine transitions to this state when one of the following events takes place: a socket connection is found to be a duplicate, an error in the socket, the peer closing the socket, and closing of the associated virtual connection.

We have written a handler function for each state of the state machine. This handler function is called by the polling loop of the progress engine for each of the socket connections when one of the requested events (such as reading or writing or both) for the socket connection has occurred. The typical work of the handler function of each state is to execute the defined action, and, if necessary, move the state machine to the appropriate state. The tasks of each of the handler functions are as follows:

- `Handler_TS_CLOSED`: This handler function gets called for both the connect-side and the accept-side of the state machines. When the virtual connection layer tries to establish a network connection, it requests the TCP module to create a connection to the remote process; the TCP module creates a socket and connects to the remote process. When this happens, the state machine of the connection moves from the `TS_CLOSED` to `TC_C_CNTD` or `TC_C_CNTING` depending on whether the socket is immediately connected to the peer or not. Similarly, when the listener socket accepts a new connection, the state machine of the accepted connection transitions from this state to `TA_C_CNTD`.

– `Handler_TC_C_CNTING`: In this state, the socket descriptor is polled for both reading and writing. If the socket becomes readable or writable and there is no error, then the state machine is moved to `TC_C_CNTD`. If there are any errors on the socket, then the state machine moves to `TS_D_QUIESCENT` state.

– `Handler_TC_C_CNTD`: We check whether there is any other connection in the connection table either in the `TS_COMMRDY` or the `TA_C_RANKRCVD` state. If such a connection is found, then we transition this socket connection to the `TS_D_QUIESCENT` state. We will see why this is done. If another connection in the table is already in the `TS_COMMRDY` state, then it is evident that this network connection need not be considered any further; it can be considered a duplicate and closed. If another connection in the table is already in the `TA_C_RANKRCVD` state, we consider that connection to be in a more advanced state already; this connection can be closed.

  If such an existing connection is not found in the table, we poll whether the socket is writable. If it is not writable, then nothing is done and the state machine continues to remain in its current state. If it is writable, then we send the identification information to the remote peer process of this connection and the state machine is transitioned to the `TC_C_RANKSENT` state. The identification information includes only the rank of the process if the remote process is in the same process group; otherwise, it includes both the rank and the process group id.

– `Handler_TC_C_RANKSENT`: The socket descriptor is polled only for its readability in this state; if it is readable, we read a packet from the socket. If we received an `ACK` packet from the peer, the virtual connection is notified that an associated network connection is ready for communication, and the state machine moves to the `TS_COMMRDY` state; if we received a `NAK` packet instead from the peer, the state machine moves to the `TS_D_QUIESCENT` state. We will explain the peer sending `ACK` versus `NAK` in the `Handler_TA_C_RANKRCVD` function. If there are any errors in the socket or the peer has closed the socket connection, then we transition from this state to the `TS_D_QUIESCENT`.

– `Handler_TA_C_CNTD`: The socket connection is checked for readability. If there are any errors in the socket, state machine goes to the `TS_D_QUIESCENT` state. If there is no error and the socket is not readable yet, we choose to remain in this state. On the other hand, if the socket becomes readable, we receive the identification information. Based on the rank and process group ID of the peer, the virtual connection is obtained and the socket connection and virtual connection are associated with each other. The state machine transitions to the `TA_C_RANKRCVD` state.

Listing IV.1 Algorithm to resolve duplicate connections

```
/*
This algorithm is executed only in the TA_C_RANKRCVD state
of the state machine of a connection. Let's assume the
algorithm is called from the state machine handler of the
connection 'A' and assume another connection called 'B'
exists already in the socket connection table that is in
TS_COMMRDY state or TC_C_RANKSENT state.
Note: PGID: Process Group ID, RANK: Rank of the process in
   MPI_COMM_WORLD
*/
procedure ResolveDuplicates()
{
  if 'B' is in TS_COMMRDY {
    'B' wins;
  }
  else if ('B' is in TC_C_RANKSENT state) {
    if (the peer of 'A' is in the same process group) {
      if (RANK of this process > RANK of the peer of 'A') {
        'A' wins;
      } else {
        'B' wins;
      }
    }
    else if (the peer of 'A' is in a different process group)
       {
      if (PGID of this process > PGID of the peer of 'A') {
        'A' wins;
      }
      else {
        'B' wins;
      }
    }
  }
}
```

– `Handler_TA_C_RANKRCVD`: The socket descriptor is polled to see whether the socket is writable. If there are any errors in the socket or it is closed by the peer, then we move it to `TS_D_QUIESCENT` state. We call the socket connection of the current state machine `A`. If there is no error in the socket and the socket is writable, then we check whether there is any connection in the connection table that is either in the `TS_COMMRDY` or the `TC_C_RANKSENT` state. If such a connection is found, we call this connection `B`. We call the algorithm `ResolveDuplicates` in Listing IV.1 and the algorithm determines whether `A` wins or `B` wins. If such a connection itself is not found in the table, then clearly `A` wins. If `A` wins, then we send an `ACK` packet to the peer and the state machine moves to `TS_COMMRDY` state; if `B` wins, then we send a `NAK` packet to the peer and the state machine moves to `TS_D_QUIESCENT` state.

– `Handler_TS_COMMRDY`: This is the state in which the associated virtual connection can send and receive MPI messages. If the socket descriptor is writable, this state handler sends the queued messages in the virtual connection; if the socket is readable, this handler receives the messages from the peer. When the virtual connection is terminated, this state machine transitions to the `TS_D_QUIESCENT` state. If there are any errors in the socket while receiving or sending, then the state machine enters the `TS_D_QUIESCENT` state as well.

– `Handler_TS_D_QUIESCENT`: If the virtual connection is terminated and messages are in the send queue and the socket is writable, those outstanding messages are sent and then the socket is closed; if there are any socket errors, or if the peer has closed its socket, then this socket is closed as well. Then, the state machine is moved to `TS_CLOSED`.

The key task our state machine performs is resolving the head-to-head situations that may occur in the on-demand connection establishment scheme. The requirement to use non-blocking sockets for the TCP network module made the state machine design more complex for the reasons explained earlier in the section. We had to be careful with some of the implementation specific details in using non-blocking sockets, as different Unix and Linux implementations have slightly different behaviors, in the areas of readability, writability, and error handling of sockets, which made it more difficult to arrive at a portable design and implementation. The original design of the state machine looked plausibly correct when manually tracing individual transition paths in the connect-side and accept-side state machines. The total machine, however, is complex enough that such "pen tests" were not sufficient to assure us of its correctness.

CHAPTER V

FORMAL VERIFICATION OF DESIGN

The old implementation of the TCP module in MPICH2 before Nemesis was introduced, called the *sock channel*, had several issues related to reliability and robustness. This was because of the complexity of the connection establishment scheme and interactions with the upper layers, the virtual connections. As the complexity of the entire system grew, the original implementation did not cater to it. Problems that occur rarely, due to the race conditions occurring in a multi-tasking or distributed environment, cannot be reproduced reliably in test laboratories. Such defects are referred to as *heisenbugs*, as there are uncertainties in trying to reproduce them. For instance, an attempt to to understand a defect by instrumenting the source code containing the defect, may cause the defect to disappear or not manifest, due to changes in timing. Such heisenbugs were primarily affecting the reliability of the sock channel.

The TCP network module — developed for the Nemesis channel in MPICH2 — was designed carefully. The implementation underwent functional testing, and system testing, however, in order to ensure the reliability and robustness of the new TCP module, we decided to formally verify the design of the state machine of the new TCP module.

A. Need for Formal Verification

Software bugs are very costly. According to a federal study done in 2002 [30], cost of software bugs to the U.S. economy was estimated to be \$59.5 billion each year, with more than half of the cost borne by end users and the remainder by developers and vendors. David Rice, the author of the book titled “Geekonomics: The Real Cost of Insecure Software,” states in the book [31] that shoddy software cost the US roughly

$180 billion in 2007 alone. Today, it is likely that software bugs cost substantially even more. The federal study [30] also reports that increasing test coverage could reduce only one third of this cost, and it will not eliminate all software errors.

It is widely understood that the cost of identifying and fixing software defects increases at later and later stages of the software development life cycle. The results of a study done by Walter Bazuik [32] and another one done by Barry Boehm [33], shown in Table II, illustrate the order of magnitude it costs to fix a software bug as a project matures.

Table II. Relative cost of fixing bugs in different stages of software development.

Life Cycle Stage	Bazuik study	Boehm study
Requirements	X	X
Design		3X–6X
Coding		10X
System Testing	90X	15X–40X
Installation Testing	90X–440X	
Acceptance Testing	440X	30X–70X
Operation and Maintenance	470X–880X	40X–1000X

Even though the software industry is aware of the high cost of fixing defects at later stages of software life-cycle, defects still slip through to these phases. For example, according to Tony Hoare, a researcher at Microsoft's Cambridge laboratory, up to three-quarters of the $400 billion spent annually to hire programmers in the United States is ultimately spent on debugging [34]. Therefore, techniques to verify the design before or during implementation (coding) stage should be used when possible.

Although testing remains the most commonly used tool to uncover software defects, testing has its drawbacks. Dijkstra says, “Program testing can be used to show the presence of bugs, but never their absence.” Ideally, it would be possible to prove the absence of defects in software programs. Indeed, theorem provers and model checkers can, in certain cases, achieve this. In this thesis, we apply model checking to ensure the correctness of the design and implementation.

A study, given in the book titled “Programming Productivity” and reproduced in Table III [35], shows the observed efficiency of different defect removal methods. We can see that *modeling or prototyping* is one of the highly efficient techniques to remove software defects.

1. Complications in Concurrent Software Verification

Concurrency (multi-tasking) makes traditional testing techniques and manual code inspection grossly inadequate to gain confidence in the correctness of even seemingly not-so-complex software systems of modest size. To see why this is, consider the number of possible thread interleavings in a few simple programs with only a few threads. Assume two threads each executing four atomic units of code. The number of different possible thread interleavings is $\binom{8}{4} \times \binom{4}{4} = 70$. Assume three threads each executing four atomic units of code. The number of possible thread interleavings in this case is $\binom{12}{4} \times \binom{8}{4} \times \binom{4}{4} = 34,650$. Finally, with three threads and eight atomic units of code, the number of thread interleavings reaches $9,465,511,770$.

As the number of lines of source code increases and the number of threads increases, the number of thread interleavings increases exponentially. Today, even simple multi-threaded software systems easily spawn dozens of threads — that span thousands of atomic units of code; therefore, the number of possible interleavings

Table III. Efficiency of several software defect removal methods.

Removal Step	Lowest(%)	Modal(%)	Highest(%)
Personal checking of design or docs	15	35	70
Informal group design reviews	30	40	60
Formal design inspections	35	55	75
Formal code inspections	30	60	70
Modeling or prototyping	**35**	**65**	**80**
Desk checking of code	20	40	60
Unit testing(single modules)	10	25	50
Function testing(related modules)	20	35	55
Integration testing(full system)	25	45	60
Field testing(live data)	35	50	65
Cumulative efficiency	93	99	99

becomes astronomical. The mapping from expressions and statements in the source language to atomic units of execution is commonly quite complex. If we add processor and operating system features such as pipelining, prefetching, cache-snooping, false cache-sharing, mutexes, semaphores, spinlocks, memory barriers, interrupt handlers, signal handers, user-kernel transitions, etc. to the mix, understanding the possible interleavings due to concurrency gets spectacularly difficult. It is thus evident that the confidence for the correctness of a software system that can be gained purely with testing and code inspection remains low.

Gerard Holzmann states that the lack of observability, controllability, and time are the main issues in testing a concurrent system [15]. It is very difficult, if not impossible, to visualize the data access patterns, process scheduling decisions, and precise interleaving of events that occur in physically distinct systems. Even if this can be done, a tester cannot control many aspects of execution in a concurrent system, for instance, thread or process interleaving — that is controlled typically only by the operating system scheduler. This leads to defects that cannot easily be reproduced, but will nevertheless manifest in a production environment. Sometimes, these environments could be *a rocket*, *Mars pathfinder* [36], or *the Large Hadron Collider* [37]. Since the number of thread interleavings grows exponentially in the concurrent software applications, even coming up with novel ideas to force the race conditions to occur in order to improve test coverage, although not practical in most cases, could take several man years, and such attempts may only turn out to be futile. The aforementioned reasons suggest that the role of formal specification and verification techniques for ensuring the correctness of concurrent programs is important, more so than for sequential programs.

B. Introduction to Model Checking

In the above section, we saw that standard testing techniques do not suffice if we desire high confidence that a software system — especially a distributed or concurrent system — is correct. One cannot prove that a system is correct in any absolute sense. One can only prove that a system does (or does not satisfy) certain specific properties [15]. There are two well-known techniques to check the correctness properties of a system.

– **Deductive proof**: The basic idea of this technique is to come up with rigorous mathematical argument — based on the axioms, proven theorems, and rules of inference — that proves a property, or absence of a property in a system.

– **Model checking**: The fundamental principle of this technique is to explore all possible system executions exhaustively, and check whether a correctness property holds.

Deductive methods — introduced by Hoare [38] — can be used to prove that the execution of a program produces an output that satisfies a property expressed in formal logic. Amir Pnueli, et al. extended these deductive systems to distributed systems [39]. While deductive methods are appreciated by mathematicians, even experienced mathematicians find it challenging to model a fairly simple distributed system. Edmund Clarke says that though he is very familiar with constructing proofs by hand using Floyd-Hoare style logic and a formal system for reasoning about *conditional critical regions*, task of proof construction is not only tedious but also makes him quite skeptical about the scalability of hand-constructed proofs [40].

On the other hand, the alternative technique is to build a simplified model of the original distributed system preserving the essential characteristics of the actual system and then verify this model. The tool with which a design model can be verified is called *model checker*, and the technique is called *model checking.* Edmund Clarke and Allen Emerson [41], the inventors of model checking, say, "Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model." Though mathematicians might consider this approach inelegant as this technique is about exploring all the possible states to prove the correctness, computer scientists consider this practical to verify a program automatically [42].

1. Model Checking and Model Checker

The formal definition of the model checking problem is the following [40]:

Let M be a Kripke structure (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that

$M, s \models f$.

The inputs to a model checker are a model of the program to verify and a correctness property. The model checker constructs two non-deterministic finite state automata — one for the model and one for the negation of the property. If it finds an input string that matches both the finite state automata, then the model checker has found a violation of the property and flags an error. If it cannot find such an input string, then the property is satisfied in the model. The challenge for the model checker is to explore exhaustively all the states of the program; this problem is known as state space explosion, and it suffers from both space and time complexity. However, model checkers have been evolving for a few decades, and the modern model checkers are smart in pruning the state space and marking the already visited states to reduce the space and time complexity, and thus making them extremely useful to check even fairly complex models.

Model checking has a number of advantages compared to other verification techniques such as the deductive proof method or automated theorem proving. Some of these advantages, according to Edmund Clarke [40], are given below:

1. The user of a model checker does not need to construct a correctness proof. In principle, all that is necessary is for the user to enter a description of the program to be verified and the specification to be checked and run the model checker tool. The checking process is automatic.

2. Using rigorous methods such as proof checkers (or theorem provers) may require months of the user's time working in interactive mode. However, it takes less time to develop a model for checking with a model checker tool.

3. If the specification is not satisfied, the model checker will produce a counterexample execution trace that shows why the specification does not hold. It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature.[1]

4. It is not necessary to specify the entire program before beginning to check the properties of a model. Thus, model checking can be used during the design of a complex system. The user does not have to wait until the design phase is complete.

5. Temporal Logics can precisely and succinctly express many of the properties that one commonly wants to prove about concurrent systems.

To verify the state machine for establishing on-demand connections developed in this thesis, we use the SPIN model checker. SPIN, acronym for "Simple Promela INterpreter," is a state-of-the-art model checker developed by Gerard Holzmann at the Bell Labs which is used not only in academia but also widely in the industry for model checking concurrent programs. SPIN was recognized with the most prestigious "Software System Award" by the ACM (Association for Computing Machinery). SPIN is also an efficient model checker and has been growing from 1991 until today with new features and bug fixes.

C. Introduction to PROMELA

PROMELA, acronym for "PROcess MEta LAnguage", is the language used to write design models that can be checked by the SPIN model checker. This section briefly

[1]We will discuss later in this chapter how this feature is used.

reviews PROMELA and SPIN; for more detailed description, we refer to the book, "The SPIN Model Checker," [15] the authoritative guide for SPIN.

1. Datatypes

PROMELA supports some primitive statements like the C language. It supports numeric datatypes such as `bit`, `bool`, `byte`, `short`, `int`, and `unsigned`. The data type `bool` and the values `true` and `false` are just syntactic sugar for the data type `bit` and the values 1 and 0. All variables, defined global or local, are initialized to 0 by default. PROMELA does not support the data types character, string, and floating point. `byte` can be used instead of the character data type and `printf` supports "%c" to print characters. The floating point and string data types are intentionally omitted as one would typically not require them in a modeling language, and numerical values are sufficient to model a system and verify it. PROMELA does not have any explicit type conversions; all values are implicitly converted to `int` for all arithmetic operations.

The datatype `mtype` can hold symbolic values; `mtype` declarations are usually placed in the beginning of the specification, and merely enumerates the symbolic names used in the model, for instance, as follows:

```
mtype = {ini, ack, dreq, data, shutup, quite, dead};
```

Because of the restriction of the value range of this type, more than 255 symbolic names cannot be declared in all `mtype` declarations combined. A special predefined routine called `printm` can be used to print the symbolic name of an `mtype` variable.

User-defined data types are also supported in PROMELA, same as C, with the keyword `typedef` as below:

```
typedef tMsg {
  mtype msg_id;
  int rank;
};
```

2. Processes

A SPIN model is commonly used to describe the behavior of a system that has concurrently executing processes. The primary unit of execution in a SPIN model is a process, and not a C-style function. The keyword `proctype` is used to specify a process:

```
active [2] proctype proces() {
  printf("Welcome to SPIN. My ID is %d\n", _pid)
}
```

The keyword `active` indicates that a process must be instantiated from the `proctype` declaration that follows, and the number two in the square brackets specifies the number of processes that must be launched. `_pid` is a reserved keyword that gives the process identifier which can be used for debugging a model. If `active` is not specified in `proctype` declaration, then the process(es) must be instantiated with the `run` operator from another process. There is a special process declared with the keyword `init` which is the first process that gets instantiated if it exists. Note that there is no ";" at the end of the `printf` statement in the above listing; semicolon is defined as a *statement separator* and not as a *statement terminator* in PROMELA.

3. Message Channels

Message channels are the language constructs that help to model the exchange of data between processes.

```
chan qname = [2] of {mtype, int}
```

In the above declaration, `chan` introduces the channel declaration; `qname` is the name of the channel that is capable of storing two messages, and each message consists of two fields, `mtype` and `int`.

The statement

```
        qname!expr1, expr2
```

enqueues a message with the values of the two expressions at the end of the channel `qname`. By default, the *send statement* is only executable if the target channel is not full; it blocks otherwise.

The statement

```
        qname?var1, var2
```

removes a message from the head of the channel and stores values from the fields of the message into the corresponding variables. The *receive statement* is executable only if the source channel is not empty.

Some or all the arguments in the receive statement can be given as constants instead of variables:

```
   qname?const1, var2
```

In the above case, only if the value of all the message fields specified as constants matches the values of the corresponding fields in the message that is to be received, the receive statement becomes executable. If we want to use the current value of the variable itself as a constant, then we can use the predefined function `eval` as follows:

```
   qname?eval(var1), var2
```

The statements that send or receive messages from a channel are called *I/O statements*.

PROMELA supports a few predefined boolean functions `full`, `empty`, `nfull`, and `nempty` that can be used to test the emptiness and fullness of the channel. `len` is a function that returns the number of messages stored in the channel.

In certain cases, we may want to check whether a send or receive operation would succeed without actually executing it. This can be achieved by changing the channel operations discussed above into side effect free expressions. For example, the expression

```
(state == CONNECTED && qname?[connect_request])
```

is true when the first condition evaluates to true and the particular receive operation can be executed; however, the actual receive operation itself is not executed when evaluating this expression. The second condition in the above statement is known as *poll* operation on the channel.

There is a special form of message communication protocol called *rendezvous message passing*, also known as *synchronous communication*. This can also be modeled using PROMELA by specifying the channel capacity as zero:

```
chan rendez_chan = [0] of {byte}
```

In other words, messages can only be passed between processes using the channel, but they cannot be stored. Therefore, rendezvous communication can happen only between two processes:

4. Rules of Executability and Control Flow

The definition of PROMELA is based on its semantics of *executability*, which is significantly different from any of the mainstream languages. In a PROMELA model of a system, every statement is either *executable* or *unexecutable* (also called *blocked*) at any given state. If and only if a statement evaluates to the boolean value `true`, that statement is executable. Otherwise, it is blocked until the statement is evaluated to `true` by the progress of the other process(es) in the system. For example, the statement

```
(state == connected);
```

blocks until `state` becomes `connected`. Since a blocking statement may have to be evaluated several times before it becomes executable, expressions in PROMELA must be free of side-effects.

The interesting and nonintuitive aspect in PROMELA is that the expressions can be used as statements in any context. For instance, when the expression

```
(state == CONNECTED && qname?[connect_request])
```

is used as a statement, the process executing the statement will block until the statement evaluates to `true`. The expressions when used as statements are referred to as *expression statements*.

An execution sequence in PROMELA is a sequence of statements in a selection or repetitive statement. The selection structure in PROMELA contains one or more possible execution sequences; each preceded by a double colon. A sequence can be selected only if its first statement is executable. First statement is therefore called the guard of the option sequence.

```
if
:: (state == COMM_READY) -> option1
:: (state != COMM_READY) -> option2
fi
```

If only one of the sequences is executable, that sequence will be executed. In the above example, only one of the guards will always be executable; however, this need not be true. If all the guards are unexecutable, the process will block until at least of one of the guards becomes executable. If more than one guard is executable, one of the sequences will be chosen non-deterministically. A special guard called `else` can be used in the selection sequence which will become executable only if none of the other guards is executable, for instance:

```
if
:: (state == CONNECTED) -> option1
:: (state == COMM_READY) -> option2
:: else -> option3
fi
```

Another statement called `skip` — that always evaluates to `true` — is a syntactic sugar for `true`. The arrow symbol “$\rightarrow$” is just another name for the semicolon that differentiates a guard from the other statements in a sequence. Otherwise, guards are simply normal PROMELA statements.

The repetitive statement is called the `do` statement:

```
do
:: (state == CONNECTED) -> option1
:: (state == COMM_READY) -> option2
:: else -> break
od
```

The `do` statement contains one or more execution sequences, each preceded by a double colon. The semantics of it is very similar to the selection structure explained above except that the control goes back to the start of the loop after executing any one of the statement sequences. For breaking out of a repetition structure, PROMELA offers the `break` statement, similar to that in C. For unconditional jumps, PROMELA supports a `goto` statement.

D. Introduction to Verification using SPIN

The correctness properties that need to be verified in a distributed system can be classified primarily into two categories: *safety* and *liveness* properties. Safety is about proving that nothing bad ever happens in the system. Liveness is about proving that something good eventually happens in the system. In other words, safety is usually designed as a set of properties that a system must not violate, while liveness is defined as a set of properties that a system must satisfy [15]. Consider the classical critical section problem where multiple processes try to enter a critical section and mutual exclusion of processes must be satisfied. A safety property may be defined as "make sure that there is *always* at most one process in the critical section." A liveness property may be defined as "if many processes try to enter a critical section, one process *eventually* enters the critical section."

1. Verification of Safety Properties

There are three ways to verify safety properties using SPIN.

– **Assertions**: Assertions in PROMELA are similar to the assertions in the mainstream programming languages. Statements of the form

  ```
  assert(num_procs_in_critsec <= 1);
  ```

  are called *basic assertions* in PROMELA. Basic assertions are always executable; and, they are side-effect free. If an `assert` statement evaluates to `false`, then SPIN triggers an error message.

– **Built-in Safety Verification of SPIN**: Some correctness properties of a distributed system need not be stated explicitly but are expected to be verified by SPIN. One such property, for instance, is that the processes in the system reach the valid end-states (i.e., processes do not deadlock or hang). SPIN's built-in safety verifier can verify that such safety properties are satisfied.

  SPIN expects all processes to reach the end of their `proctype` body (the final curly brace) by default. However, some processes (for example, server processes) do not reach the end of the `proctype` body but are possibly in a wait-state in the loop (for example, waiting to service a client) — which is correct by design and must not be considered an invalid end-state by SPIN. Therefore, PROMELA allows specifying user defined end-states (through the use of labels that are prefixed with "end") that are honored by SPIN as valid end-states.

– **LTL and *never* claims**: There is often a requirement to verify that some invariants are satisfied in all possible states. These invariants can involve more than one process. One way to verify such invariants using SPIN is to have an additional process that asserts the invariant. For example, in the code snippet shown in Listing V.1 where two processes try to enter and leave a critical section, a `Watchdog` process is created to make sure that the number of processes in the critical section is always less than or equal to one.

Listing V.1 Critical section.

```
int lock = 0;
int num_procs_in_cs = 0;

active proctype P()
{
l1:
  do
  ::
      enter_cs(lock);
  l2:
      num_procs_in_cs++;
      num_procs_in_cs--;
      leave_cs(lock);
  l3:
      skip;
  od
}

active proctype Q()
{
  do
  ::  enter_cs(lock);
      num_procs_in_cs++;
      num_procs_in_cs--;
      leave_cs(lock);
  od
}

active proctype Watchdog()
{
  assert(num_procs_in_cs <= 1);
}
```

Though the above technique with a special process works to verify system invariants, SPIN allows a more elegant way to verify the properties using `never` claims. A `never` claim is normally used to specify a finite or infinite system behavior that should never occur [15]. A `never` claim can be thought of a special process that checks a property before and after each execution step of the entire system. For verifying an invariant like the one defined in the `Watchdog` process in Listing V.1, the special claim process simply checks for the invariant in every step of execution and flags an error if the property is violated in any step of the execution. The code snippet below shows the `never` claim that can obviate the `Watchdog` process.

```
#define p (num_procs_in_cs <= 1)

never n1 { /* ![] p */
start:
  if
  :: (1) -> goto start
  :: (!p) -> goto accept_all
  fi;
accept_all:
  skip
}
```

SPIN flags an error if the `never` claim reaches a state that is labeled with the prefix `accept`.

`Never` claims for complex properties are not easy and intuitive to write; however, the complex properties can be easily written as claims in *Linear Temporal Logic* (LTL). For example, the claim `sf1` says that `p` must be *always* true. The symbol `[]` is read as *always* in LTL. We postpone the discussion of LTL itself to Section 3.

Listing V.2 Safety property in critical section.

```
#define p (num_procs_in_cs <= 1)
ltl sf1 { [] p }
```

2. Verification of Liveness Properties

While verifying safety properties of a logic model is important, safety properties could simply be vacuously true in a model that does not do anything useful. For instance, in the critical section problem — shown in Listing V.1 — the safety property using `Watchdog` process, the `never` claim `n1`, or the LTL claim `sf1` could simply be satisfied in an erroneous solution where no process ever enters the critical section. Therefore, we need to verify *liveness* properties of a system. Liveness properties can be written using `never` claims or LTL claims. We show a few liveness properties in LTL for the critical section problem in Listing V.3.

Listing V.3 Liveness properties in critical section.

```
#define try_cs (P@l1)
#define in_cs (P@l2)
#define out_of_cs (P@l3)

ltl lv1 { [] (try_cs -> <> in_cs }

ltl lv2 { [] (in_cs -> <> out_of_cs) }
```

"`<>`" is read as *eventually* and "`->`" is read as *implies.* The LTL claim `lv1` states that "If process `P` tries to enter a critical section, it eventually does so"; the LTL claim `lv2` states that "If process `P` is in a critical section, it eventually leaves the critical section." To appreciate the brevity of LTL claims over `never` claims (and ease of writing LTL claims), we show the `never` claim for the LTL property `lv1` in Listing V.4.

Listing V.4 never claim for property lv1.

```
never  {    /* [] (p -> <> q) */
T0_init:
  if
  :: (((! ((p))) || ((q)))) -> goto accept_S20
  :: (1) -> goto T0_S27
  fi;
accept_S20:
  if
  :: (((! ((p))) || ((q)))) -> goto T0_init
  :: (1) -> goto T0_S27
  fi;
accept_S27:
  if
  :: ((q)) -> goto T0_init
  :: (1) -> goto T0_S27
  fi;
T0_S27:
  if
  :: ((q)) -> goto accept_S20
  :: (1) -> goto T0_S27
  :: ((q)) -> goto accept_S27
  fi;
}
```

3. Linear Temporal Logic

Temporal Logic is a branch of logic that extends propositional calculus [43] with some temporal operators. This logic allows the formalization of safety and liveness properties, and is thus useful for verifying the correctness of a distributed system. There are several forms of the temporal logic; SPIN uses the one called Linear Temporal Logic (LTL) [15, 44]. The operators supported in LTL are given in Table IV.

LTL operators operate on the propositional symbols. The propositional symbols in LTL are boolean expressions that can be evaluated in any state of the verification of the model. For instance, `p` in Listing V.2 and `try_cs`, `in_cs`, and `out_cs` in Listing V.3 are propositional symbols.

Even though LTL claims are much easier to write than `never` claims, LTL claims are less expressive than `never` claims (i.e., all `never` claims cannot be expressed as LTL claims, but the converse is true); however, most of the interesting properties can indeed be specified in LTL for most of the model checking use cases. SPIN internally

Table IV. LTL operators.

Operators	Math	SPIN
not	$\neg$	`!`
and	$\wedge$	`&&`
or	$\vee$	`||`
implies	$\rightarrow$	`->`
equivalent	$\leftrightarrow$	`<->`
always	$\Box$	`[]`
eventually	$\Diamond$	`<>`
until	$\cup$	`U`

converts LTL claims to `never` claims before verification.

4. Model Abstraction

SPIN not only supports design abstractions, it requires them [15]. The purpose of SPIN is to verify concurrent applications. In a PROMELA model, the focus is on the control aspects of the applications, not the computational aspects. In order to make sure that the models specified in PROMELA always have effectively verifiable properties, two requirements are imposed [15]:

- the model can specify only finite systems, though the underlying application is potentially infinite.
- the model must be fully specified, that is, it must be closed to its environment.

A program that allows unbounded recursion is not finite. A program that reads input from a file or stream is not closed to its environment. For most of the practical software applications, the aforementioned conditions are not automatically met. Therefore, we have to apply abstraction to construct SPIN models. Holzmann recommends the following steps to develop a verification model [15]:

- The aspects of the design — that are important and require verification — should be determined, and should be expressed as a set of system requirements. The requirements must be *testable*.
- The essence of the design itself — specifically the aspects of the design that help the system meet its requirements — should be considered.
- An executable abstraction in PROMELA, also known as the model, should be constructed. The model should be detailed enough to capture the essence of the design or implementation, and no more.

The verification model must allow us to make refutable statements about the design. Therefore, aspects of a model that do not contribute to refutability can and should be deleted to enhance the verifiability [15]. Holzmann further stresses the need for building a smallest sufficient model using efficient abstraction because the computational complexity remains the single most challenging issue for model checkers, even though the model checkers have evolved significantly, and computers have got orders of magnitude faster in the past two decades [15].

5. Property-based Slicing

We briefly review the definition of program slicing as given by Frank Tip [45] here before discussing SPIN's slicing techniques: A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion* and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing*. Details on different types of program slicing and different techniques and algorithms in program slicing can be found in [45].

Holzmann's informal definitions of logical completeness and logical soundness of an abstraction are as follows [15]: "An abstraction is *logically sound* if it excludes the possibility of false positives. The correctness of the model always implies the correctness of the program. An abstraction is *logically complete* if it excludes the possibility of false negatives. The incorrectness of the model always implies the incorrectness of the program". The formal definitions of logical soundness and logical completeness can be found in [15].

An abstraction method that guarantees both logical soundness and logical completeness with respect to a given LTL property is called *selective data hiding* [15]. For using this method, a set of data objects that are irrelevant to the property — that needs to be proved — should be identified and removed from the model together with all the associated operations on those data objects. Using a simple version of the *program slicing* algorithm, this abstraction method is automated in SPIN.

The simple version of program slicing algorithm built into SPIN does the following [15]: a set of *slice criteria* is constructed including only the data objects that are referred to explicitly in one or more correctness properties defined using either assertions or LTL formula. Through data and control dependency analysis, the algorithm determines on which larger set of data objects the slice criteria depend for their values. All data objects that are independent of the slice criteria, and not contained in the set of slice criteria themselves, can then be considered irrelevant to the verification and thus can be removed from the model, together with all associated operations. This technique is referred to as *property-based slicing*.

Applying the property-based slicing on a model, SPIN reports two types of feedback about the model:

– **Redundancies in the model:** Property slicing algorithm in SPIN reports the redundancies (lines of code and variables) found in the model for the given LTL property or properties (and assertions) in the model. If redundancies were found in the model, it helps with verification in different aspects. If a model is too complex to verify due to constraints such as time or memory usage, then the model can be simplified by removing redundant variables and code for the given property and the simplified model can be verified for the given property. This can be repeated for different properties that need to be verified.

  If the same lines of source code are reported as redundant by the property slicing algorithm for all the properties that need to be verified in a model, then it could imply that the model could be simplified by removing the redundant lines of code. However, there is a caveat: the redundant lines of source code reported by SPIN are not *provably completely irrelevant* and thus cannot be automatically removed without manually examining them and ensuring their necessity. The reason is the following: Although, property-based slicing can be shown to preserve both logical soundness and logical completeness of the correctness properties that are used in deriving the abstraction, it does not necessarily have these desirable properties for some other types of correctness requirements that cannot be expressed in assertions or LTL formulae. An example of such a property is "absence of deadlock" [15]. Therefore, the designer should be careful about removing redundant lines of code in the model; the original model can suffer from deadlocks even though the simplified model does not.

– **Less-restrictive data types:** SPIN recommends using less-restrictive data types, if possible. For instance, SPIN recommends using `byte` instead of `int`. Less-restrictive data types help with reducing verification complexity; for instance, the size of the state vector is reduced.

E. Details of Verification of the Model

The key aspect in the design of the on-demand connection establishment scheme is the state machine explained in Chapter IV. The state machine handlers executing in two different processes can interleave in several ways, and thus formally verifying the state machine is essential to gain confidence about the design. Therefore, we construct a model of the state machine in PROMELA and verify various safety and liveness properties using SPIN.

1. Extraction of the Model

Abstracting the model from the design is the foundation to "model checking". A poor abstraction may either cause frustration during the verification of properties or may give false confidence about the success of verification. Therefore, we carefully abstract the PROMELA model from the design of the state machine. As a first step in abstraction, we map the key aspects of the design to the idioms and constructs of PROMELA and ignore the other details — that are not relevant to the verification of the design.

We discussed network programming with sockets in Chapter II and usage of non-blocking TCP sockets in Chapter IV. The closest PROMELA idiom to exchange data back and forth is channels. Since the network socket is full-duplex (as discussed in Chapter II), we need two channels for modeling one network socket. We define a socket using `typedef` and array of two channels (with a capacity of two messages) as follows:

```
typedef tSocket {
  chan ch[2] = [2] of {tMsg};
};
```

In the code snippet in Listing V.5, we show how a server (that accepts a socket connection) and a client (that initiates a socket connection), such as the program in Listing II.1, can be modeled in PROMELA.

Listing V.5 TCP client-server using channels.

```
proctype client(chan con_snd, con_rcv) {
}

proctype server(chan acpt_rcv, acpt_snd) {
}

init {
  tSocket sock;

  client(sock.ch[0], sock.ch[1]);
  server(sock.ch[0], sock.ch[1]);
}
```

The send-channel of the "connecting" socket — `con_snd` — in the `client` is used as the receive-channel of "accepting" socket — `acpt_rcv` — in the `server`; similarly, the send-channel of the "accepting" socket — `acpt_snd` — in the `server` is used as the receive-channel of "connecting" socket — `con_rcv` — in the `client`. The variable `sock` itself cannot be passed as an argument to the `client` and the `server` directly due to the limitation of PROMELA not allowing either an array or a `typedef` containing an array to be passed as an argument to `proctype`. PROMELA also requires that `typedef` must be defined globally and not in the process context.

We reviewed in Chapter IV that, in the on-demand connection scheme, each

MPI process can initiate a connection to another MPI process and can accept a socket connection from another MPI process simultaneously. Therefore, we model two processes each trying to connect to the other and accept connection from each other. Both the processes execute the same state machine in the MPICH2 implementation. Therefore, we define one process called `NetModSM` to represent the state machine of the TCP network module of the Nemesis channel, and instantiate two instances of this process as given in the following listing:

```
typedef tSocket {
  chan ch[2] = [2] of {tMsg};
};

proctype NetModSM(byte id; chan con_snd, con_rcv, acpt_rcv,
   acpt_snd) {
}

init {
  tSocket sock[2];
  byte proc_id = 0;

  run NetModSM(proc_id, sock[0].ch[0], sock[0].ch[1], sock
     [1].ch[0], sock[1].ch[1]);
  proc_id++;
  run NetModSM(proc_id, sock[1].ch[0], sock[1].ch[1], sock
     [0].ch[0], sock[0].ch[1]);
}
```

As shown in the above listing, we pass a unique ID called `proc_id` as the first argument to the process `NetModSM`.

The messages that are exchanged between two processes and the different states of the state machine (discussed in Chapter IV) are defined using `mtype` as follows:

```
mtype = {m_close, m_connect, m_connect_ack, m_rank,
   m_rank_ack, m_rank_nak};

mtype = {TS_CLOSED, TC_C_CNTING, TC_C_CNTD, TC_C_RANKSENT,
   TC_C_RANKRCVD, TS_COMMRDY, TS_D_QUIESCENT, TA_C_CNTD,
    TA_C_RANKRCVD};
```

Message exchanged between processes is defined using the `typedef` in Listing V.6.

Listing V.6 Type definition of the message.

```
typedef tMsg {
  mtype msg_id;
  int rank;
};
```

The field `rank` is included in the `typedef` and hence it is sent and received in every type of message; however, `rank` is relevant only in the message type `m_rank` (as mentioned in the state machine handlers in Chapter IV). Adding an extra field — that is not needed for messages — does not increase the state-space of the verification. The other approach would be to send only the `msg_id` (such as `m_connect`) for all the messages and send another extra message with the rank information only for the `m_rank` message. This alternative approach adds unnecessary complexity to the model and increases the verification complexity (increases the state space); in addition, this would not accurately represent the actual design and implementation where a single message type is used for `m_rank`. This is an example where adding an unused field in the message for various types of messages in the model is appropriate in abstracting the PROMELA model. The downside of this approach is an increase in the size of the state-vector used by SPIN — which does not pose any practical limitations on the memory usage of the verification of our model.

As we are modeling two processes each accepting connections and initiating connections, we need to represent the state of both "connecting" socket and "accepting" socket in both processes. This is defined using an array of `mtype` — as shown below in the code snippet — where array index represents the unique `proc_id` that is passed to `NetModSM`. Both the "connecting" and "accepting" socket states of both processes are initialized to the `TS_CLOSED` state:

```
mtype con_state[2] = TS_CLOSED, acpt_state[2] = TS_CLOSED;
```

We discussed in Chapter II, in the MPICH2 implementation, how a virtual connection at the CH3 layer requests the TCP network module layer to connect to and disconnect from the remote process. We model these virtual connection and disconnection requests, sent from the `init` process to the state-machine process `NetModSM`, using channels. The snippet below shows the channels and the virtual connection (VC) messages:

```
mtype = {m_vc_connect, m_vc_disconnect};
chan vc_chan[2] = [2] of {mtype};
```

In the previous few listings, we have used the datatype `mtype` to define messages for the virtual connection and the socket connection, and various states of the "connecting" and "accepting" sockets. This reduces the readability of the model as different types of constants are defined with one datatype `mtype`. User-defined datatypes can be used to solve this problem. A `typedef` declaration, however, cannot be used to define a new datatype for an existing datatype as done in "C"; for instance, the following is not allowed in PROMELA:

```
typedef mtype tVCmsg;
tVCmsg = {m_vc_connect, m_vc_disconnect};
```

Instead, macros in PROMELA can be used to represent user-defined datatypes. We rewrite various messages and states using macros — to improve readability — as follows:

```
#define tVCmsg mtype
#define tNetMsg mtype
#define tNetModState mtype
```

```
tNetModState = {TS_CLOSED, TC_C_CNTING, TC_C_CNTD,
   TC_C_RANKSENT, TC_C_RANKRCVD, TS_COMMRDY, TS_D_QUIESCENT,
    TA_C_CNTD, TA_C_RANKRCVD};

tNetModState con_state[2] = TS_CLOSED, acpt_state[2] =
   TS_CLOSED;

tVCmsg = {m_vc_connect, m_vc_disconnect};

chan vc_chan[2] = [2] of {tVCmsg};

tNetMsg = {m_close, m_connect, m_connect_ack, m_rank,
   m_rank_ack, m_rank_nak};

typedef tMsg {
  tNetMsg msg_id;
  byte rank;
};

typedef tSocket {
  chan ch[2] = [2] of {tMsg};
};

tSocket sock[2];
```

In the algorithm in Listing IV.1, both `PGID` (process group ID) and `RANK` are used in resolving the duplicate connections. However, taking a closer look at the algorithm, we can observe that it is sufficient to verify this algorithm only with `RANK`; `PGID` is an implementation detail that does not allow us to make any additional refutable statements (verification claims) about the model (than `RANK` by itself) and thus it does not contribute to the verifiability of the model. Therefore, we have only included `rank` in the `typedef tMsg` as shown in Listing V.6.

We have shown code snippets and illustrated some abstraction aspects of the PROMELA model from the design and implementation of the state machine. The complete PROMELA model of the state machine is given in Appendix A. The entire state machine is abstracted in the `NetModSM` process within a `do` loop with state handlers as different selection options of the loop. We use the `init` process to instantiate two instances of `NetModSM`, and then to send the virtual connection messages (`m_vc_connect` and `m_vc_disconnect`) to the `NetModSM` processes. As we discuss further about different verification aspects of the model, we recommend that the reader refers to the model in Appendix A.

2. Verification of Safety Properties

In this section, we discuss how we use the three different methods (discussed in Section D) to verify the safety properties.

– **Assertions**: Assertions are added to ensure that a channel is not full before sending a message to the channel.

```
assert(nfull(ch));
ch!msg;
```

When the “connecting” socket of a process goes to the `TS_COMMRDY` state, the “accepting” socket of the process must not be in the `TS_COMMRDY` state, and vice versa. These conditions are asserted as follows:

```
::  (con_state[id] == TS_COMMRDY &&
    (vc_chan[id]?[m_vc_disconnect] || nempty(con_rcv)))
       ->
      assert(acpt_state[id] != TS_COMMRDY);
      ...


::  (acpt_state[id] == TS_COMMRDY &&
    (vc_chan[id]?[m_vc_disconnect] || nempty(acpt_rcv)))
        ->
      assert(con_state[id] != TS_COMMRDY);
      ...
```

Even though these assertions are helpful for a sanity check during the development of the model, they are not sufficient to prove the correctness because their scope is limited to the specific states they are used (i.e., these assertions are not evaluated in all the states during verification).

– **Built-in Safety Verification of SPIN**: The model is verified for the built-in safety properties — such as invalid end-states. In the state machine resolving head-to-head resolution of two MPI processes, deadlocked state — where each process would be expecting a particular message from the other process causing both of them not to make progress — is an uncommon error. SPIN verifying the model for deadlock states without any additional user effort (such as writing claims) is a powerful feature that we exploit.

– **LTL and *never* claims**: The algorithm in Listing IV.1 is developed to solve the head-to-head resolution problems discussed in Chapter III. We need to verify formally that simultaneous head-to-head connections between two processes are resolved such that two connections between a pair of processes do not go to the `TS_COMMRDY` state. Though this property can be verified using an additional `Watchdog` process as discussed in Section D, we skip that technique here and discuss how this property can be verified using more elegant techniques such as the `never` claim and the LTL claim.

  First, we define some of the commonly used boolean conditions — whether the "connecting" socket or the "accepting" socket is in the `TS_COMMRDY` state or the `TS_CLOSED` state — as propositional symbols, which are useful to write the claims:

```
#define p (con_state[0] == TS_COMMRDY)
#define q (acpt_state[0] == TS_COMMRDY)
#define r (con_state[1] == TS_COMMRDY)
#define s (acpt_state[1] == TS_COMMRDY)

#define p_ (con_state[0] == TS_CLOSED)
```

```
#define q_ (acpt_state[0] == TS_CLOSED)
#define r_ (con_state[1] == TS_CLOSED)
#define s_ (acpt_state[1] == TS_CLOSED)
```

In SPIN, the `never` claim is meant to match the behavior that should never occur. We show the `never` claim below to assert that both the "connecting socket" and the "accepting socket" (of either of the two processes) do not move to the `TS_COMMRDY` state:

```
never n0 {
  do
  ::  ((p && q) || (r && s)) -> break
  ::  else
  od
accept_all:
}
```

If either `(p && q)` or `(r && s)` is satisfied, the `never` claim breaks out of the loop, and the claim reaching a label prefixed with `accept` is identified as a violation by SPIN during verification. If neither of these conditions becomes true, then the `never` claim does not exit out of the loop; this is considered successful verification of the claim.

The never claim `n0` can be written as a LTL claim as follows:

```
ltl p0 {
 [] ((p -> !q) && (q -> !p) && (r -> !s) && (s -> !r))
}
```

`(p → !q)` reads "connecting socket of `process-0` in the `TS_COMMRDY` state *implies* that the 'accepting socket' of the same process is *not* in the `TS_COMMRDY` state". As a reminder, the symbol "`[]`" is read as *always.* The claim above asserts that all the four implications are always true. `(q → !p)` is the contra-positive of `(p → !q)`. In the propositional logic theory, a statement and its contra-positive are equivalent [46]. Therefore, the above claim can be simplified as follows:

```
ltl p0 {
 [] ((p -> !q) && (r -> !s))
}
```

This LTL claim is simpler and more intuitive (as there is no negation of the property as in the `never` claim) than the `never` claim to understand (and to write). Therefore, in the rest of this Chapter, we use only LTL claims.

3. Verification of Liveness Properties

We verified, in the previous section, the safety property that both the “connecting socket” and the “accepting socket” do not go to the `TS_COMMRDY` state at the same time in a process. That property could be satisfied in a flawed model where neither socket connection goes to the `TS_COMMRDY` state. Therefore, verifying liveness properties such as “one of the socket connections in each of the two processes eventually goes to the `TS_COMMRDY` state” is a necessary part of a proof of the correctness of the model. We review some of the liveness properties we verified in this section.

We developed a few test cases that contain typical situations/scenarios that are found in real MPI applications and wrote LTL claims for those cases. The simplest case is that an `m_vc_connect` is sent to one of the two `NetModSM` processes. In this case, the “connecting socket” of the process — to which `m_vc_connect` is sent — and the “accepting socket” of the other process must go to the `TS_COMMRDY` state.

Listing V.7 Verification of VC-Connect in one process.

```
init
{
  vc_chan[0]!m_vc_connect;
}

ltl p1a {
  [] (vc_chan[0]?[m_vc_connect] ->  <> (p && s && r_ && q_))
}

ltl p1b {
  [] <> (p && s && r_ && q_)
}
```

The LTL claim `p1a` in Listing V.7 reads: It is *always* true that if an `m_vc_connect` message is found in the `vc_chan` in process-0, then the propositional state formula `(p && s && r_ && q_)` *eventually* becomes true. The formula indicates that the "connecting socket" of process-0 and "accepting socket" of process-1 go to the `TS_COMMRDY` state, and, the "connecting socket" of process-1 and the "accepting socket" of process-0 go to the `TS_CLOSED` state. Since the claim `p1a` is written for the specific test case, it can be re-written as `p1b` in Listing V.7, omitting the conditional part of the *implication* (i.e., `(vc_chan[0]?[m_vc_connect])`).

The important scenarios for which we want to verify the liveness properties in the model are the head-to-head situations. For instance, when an `m_vc_connect` message is sent to both the `NetModSM` processes, only one socket in each process should go to the `TS_COMMRDY` state, and other other socket should go to the `TS_CLOSED` state; and the corresponding sockets in both the processes (the "connecting socket" in one process corresponds to the "accepting socket" in the other process) must go to the same state. The test case and the claims are shown in Listing V.8.

Listing V.8 Verification of VC-Connect in both the processes.

```
init {
  vc_chan[0]!m_vc_connect;
  vc_chan[1]!m_vc_connect;
}

ltl p3a {
  [] <>((p && s && r_ && q_) ||  (r && q && p_ && s_))
}
```

```
ltl p3b {
  ([] <> (r && q && p_ && s_)) || ([] <> (p && s && r_ && q_
     ))
} // equivalence of p3a
```

The claim p3a in Listing V.8 reads: It is always true that eventually either "connecting socket" of process-0 and "accepting socket" of process-1 go to the TS_COMMRDY state (and "connecting socket" of process-1 and "accepting socket" of process-0 go to the TS_CLOSED state) or "connecting socket" of process-1 and "accepting socket" of process-0 go to the TS_COMMRDY state (and "connecting socket" of process-0 and "accepting socket" of process-1 go to the TS_CLOSED state). In different possible execution sequences depending on the relative progress of one process over the other, either the state formula (p && s && r_ && q_) or (r && q && p_ && s_) will eventually become true. LTL claim p3b in Listing V.8 is simply an equivalence of LTL claim p3a, and we show that this claim can be specified as p3a or p3b.

Next, we show a case where an m_vc_disconnect is issued to process-0 after both processes move to the TS_COMMRDY state (due to a previous m_vc_connect sent to process-0):

```
init
{
  vc_chan[0]!m_vc_connect;
  (p && s && r_ && q_);
  vc_chan[0]!m_vc_disconnect;
}

ltl p4a { [] <> ((p_ && s_ && r_ && q_)) }
```

The claim p4a shown above for this case verifies that both the socket connections in both the processes move to the TS_CLOSED state eventually. We show a few more test cases and the corresponding LTL claims for those cases in Appendix A.

4. Experiences on Finding and Fixing Defects Using SPIN

Checking the model we developed for the safety and liveness properties described above helped us to discover defects in our implementation, and in our model. We discuss in this section our experience with finding one of the bugs and fixing it.

In the prototype of on-demand connection establishment scheme developed in C and the model written in PROMELA, the first `if` statement in the algorithm defined in Listing IV.1 was omitted by mistake. In the functional testing of C implementation, no defects were discovered. SPIN was very useful for incrementally developing the model and verifying the specific claims written thus far. Listing V.7 and Listing V.8 show the code for the model and the claims at different stages of development. For the test case in Listing V.7, where `m_vc_connect` is sent to only one `NetModSM` process, we found no verification errors for the claims `p1a` and `p1b`. However, when the model was further developed, as in Listing V.8 (where `m_vc_connect` is issued to both the `NetModSM` processes) and verified, the verification claim `p3a` failed and created an error trail. We replayed the error trail using the simulation mode of SPIN to understand the error in the model.

We used the graphical front-end tool called *xSpin* (and the newer version called *iSpin*) to replay the error trail. One window pane of iSpin shows the complete path of all the processes, including state number and line number of the model as below:

```
132:  proc  1 (NetModSM) sm.pml:90 (state 149)  [acpt_snd!
   acpt_msg.msg_id,acpt_msg.rank]
132:  proc  1 (NetModSM) sm.pml:91 (state 150)  [acpt_state[
   id] = 12]
134:  proc  2 (NetModSM) sm.pml:177 (state 50)  [(((
   con_state[id]==TC_C_RANKSENT)&&nempty(con_rcv)))]
136:  proc  2 (NetModSM) sm.pml:178 (state 51)  [con_rcv?
   con_msg.msg_id,con_msg.rank]
138:  proc  2 (NetModSM) sm.pml:182 (state 54)  [((con_msg.
   msg_id==m_rank_ack))]
140:  proc  2 (NetModSM) sm.pml:183 (state 55)  [con_state[
    id] = TS_COMMRDY]
<<<<<START OF CYCLE>>>>>
spin: trail ends after 142 steps
#processes: 3
142:  proc  2 (NetModSM) sm.pml:122 (state 187)
142:  proc  1 (NetModSM) sm.pml:122 (state 187)
142:  proc  0 (:init:) sm.pml:421 (state 6)
```

Another window pane shows the values of the global variables as listed below:

```
acpt_state[0]  =  TS_COMMRDY
acpt_state[1]  =  TS_COMMRDY
con_state[0]  =  TS_COMMRDY
con_state[1]  =  TS_COMMRDY
```

The other nice feature of iSpin is the ability for viewing values of variables, contents of channels, and states of processes, during replay of error trails. From the output of the final state of all the variables, we can see how the claim p3a defined in Listing V.8 is violated. Replaying step-by-step and looking at the model at the specific line numbers of each state, we can easily see what sequence of interleavings of the two processes leads to the error. xSpin/iSpin has a feature called *Message Sequence Charts*(MSC); messages sent and received across channels between processes are shown graphically with time-line. An example of playing an error tail is shown in Figure 9. Clicking mouse on the different boxes updates all the other windows such as variable window, error trail window, and source code listing window. The MSC feature is extremely useful for understanding errors.

Using the aforementioned techniques, the problem — why the LTL claim p3a failed — was not hard to spot and reason about. Adding the code snippet (that moves the "accepting" socket to the TS_D_QUIESCENT state if the "connecting" socket in the given process is already in the TS_COMMRDY state) to the guard statement for the TA_C_RANKRCVD state, as given in the listing below, fixed the error in the model:

```
  ::  acpt_state[id] == TA_C_RANKRCVD ->
...
        ::  con_state[id] == TS_COMMRDY ->
```

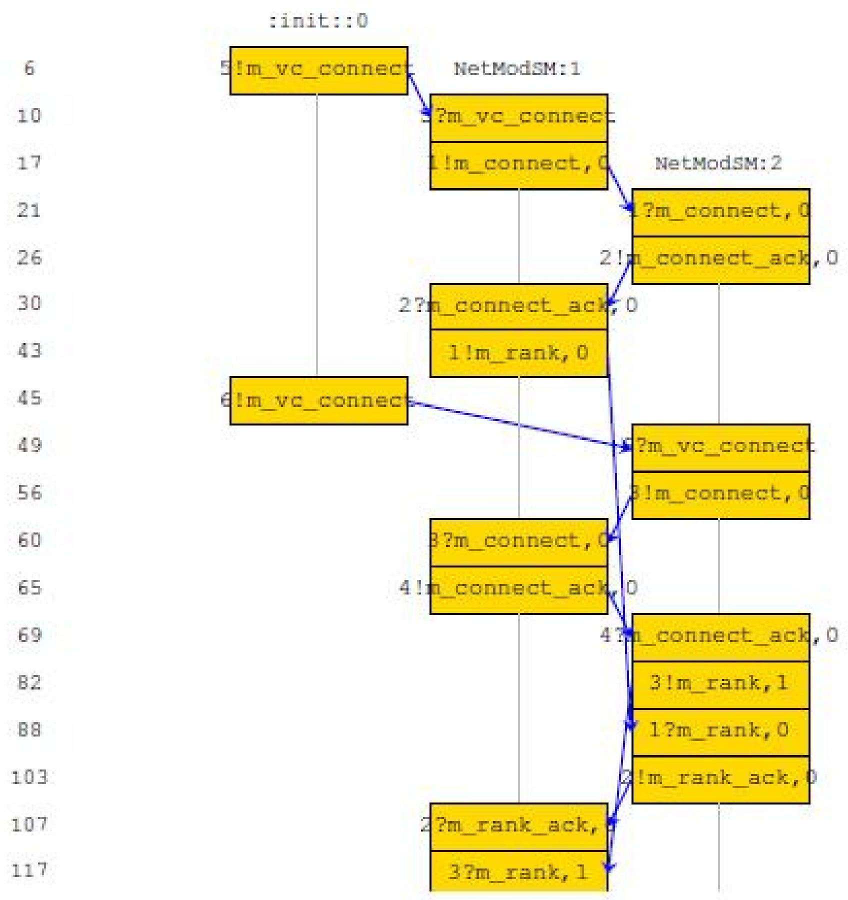

Fig. 9. Message Sequence Chart (MSC) of an error trail replay.

```
                acpt_msg.msg_id = m_rank_nak;
                send_msg_trans(acpt_snd, acpt_msg,
                    acpt_state[id], TS_D_QUIESCENT);
...
```

5. Checking for Redundancies with Property-based Slicing Techniques

We ran the property slicing algorithm of SPIN on our PROMELA model. SPIN did not report any redundancies for the LTL properties and assertions. Therefore, we did not have to do any further analysis on redundancies.

SPIN reported a few errors similar to the one shown below:

```
  spin: consider using predicate abstraction to replace:
    int   x 0 <NetModSM>  <variable>  {scope _6_}
```

The datatypes of the reported variables could be defined as more restrictive datatypes (smaller in size). In the above example, x can be defined as `byte` or even as `bit` which reduces the verification complexity of SPIN. We fixed a few errors like the above. We also removed some unused variables.

Changing the datatype of `proc_id` and `rank` from `int` to `byte` reduced the size of the state vector by 100 bytes and total memory usage of SPIN during verification by a few megabytes. Even though it is not significantly important for verification of our model, it is good practice to follow the programming idioms and best practices of the modeling language.

6. Complexity of Verification

In this section, we review the verification complexity of the model of the state machine. We discussed, in the previous sections, how we developed different test cases and verified different safety and liveness properties using LTL claims for each of the test cases. In this section, we analyze the complexity of verification for three sample test cases that are listed in Table V.

Table V. Description of test cases.

Test	Description of the test
CASE-1	vc_connect in one process
CASE-2	vc_connect in both processes
CASE-3	both vc_connect and vc_disconnect in both processes

The complexity results of verification of a safety property are shown in Table VI, and the complexity results of a liveness property are shown in Table VII.

Table VI. Verification complexity of a safety property.

Test-case	States stored	State transitions
CASE-1	75	101
CASE-2	3,267	5,523
CASE-3	21,884	43,507

As can be seen in Table V, **CASE-1** is the test case with no head-to-head resolutions (and thus the simplest one), and **CASE-2** is the test case that involves head-to-head resolutions. The number of state transitions increases from 101 for **CASE-1** to 5,523 for **CASE-2** for the safety property; the same increases from 363 to 25241 for the liveness property. We can see how the complexity of the state machine is multiple orders of magnitude greater for the head-to-head resolution compared to a no head-to-head situation. The complexity increases even more when a disconnection sequence follows the connection sequence for **CASE-3** in both safety and liveness

Table VII. Verification complexity of a liveness property.

Test-case	States stored	State transitions
CASE-1	146	363
CASE-2	6,257	25,241
CASE-3	43,667	194,834

cases. Such a huge number of states and state transitions of verification indicates clearly why it is difficult for developers to verify all the possible race conditions with code inspection, peer reviews, and traditional testing in distributed computing.

Looking at the state machine depicted in Figures 6, 7, and 8, even though the state machine is not trivial, it does not look significantly complex either. However, looking at the number of state transitions shown in the Tables VI and VII, we infer that the complexity of the state machine indeed should not be underestimated.

CHAPTER VI

BENCHMARK RESULTS

One of the motivations for developing the on-demand connection establishment scheme, as discussed in Chapter III, was to reduce the time that `MPI_Init` took in the original implementation of Nemesis's TCP module — which had to establish $\Theta(N^2)$ connections. In our on-demand connection establishment scheme, `MPI_Init` should take very little time, as it does not establish any connections. We confirmed this by benchmarking the time taken by `MPI_Init` of the original static connection establishment scheme and our on-demand connection establishment scheme; we ran our benchmarks on two desktop computers and a cluster computer. This chapter explains the benchmarking strategy and reports the results.

A. Benchmark of `MPI_Init` Time

We wrote an MPI program that calls `MPI_Init` and takes time-stamps before and after this function call. We use the Linux system call `gettimeofday` that gives us resolution in microseconds. We use `MPI_Reduce` to compute the maximum of the difference between the two time-stamps from all the processes at process-0 (that is, process whose rank is zero). In order to measure the time taken by `MPI_Init` accurately, we ignore the time to launch all the processes by calling the `PMI_Barrier` function before recording the first time-stamp. `PMI_Barrier` function is not a standard MPI function, but rather an MPICH2 implementation function. The standard MPI barrier function — `MPI_Barrier` — cannot be used, since that function can be called only after `MPI_Init`.

We use the same project manager, *Hydra*, for both static and on-demand scheme implementations in Nemesis to ensure that we precisely measure only `MPI_Init` and remove an additional variable in the benchmarking mechanism. We use the environment variable `MPICH_NO_LOCAL` to force using the TCP network module between processes in the same node also (Otherwise, shared memory would be used for intra-node communication).

We ran the benchmark program with static connection support and on-demand connection support on two desktop PCs with the following configuration. These two computers are in the same subnet (i.e., connected to a router/switch). Since there are only 6 CPU cores (ignoring hyper-threaded cores) in both the computers together, the cores are oversubscribed (that is, each processor core running more than one MPI process) when we launch more than 6 processes. The results are shown in Figure 10.

Configuration of Computer-1

Processor : Intel Core i5 2.27 GHz

Number of CPUs : 2

Number of CPU threads : 4 (with Hyperthreading)

Memory : 4 GB

OS : Linux kernel 3.0.0-12

Configuration of Computer-2

Processor : Intel Core2 Quad Q9300 2.5 GHz

Number of CPUs : 4

Number of CPU threads : 4 (no Hyperthreading)

Memory : 3 GB

OS : Linux kernel 3.0.0-12

We also ran the benchmark program on a cluster system called *Fusion* at *Laboratory Computing Resource Center* in Argonne National Laboratory. Its configuration is given below. Each node has two quad-core CPUs; eight processes per node are chosen to make sure that no processor core is oversubscribed. The benchmark results are depicted in Figure 11.

Configuration of Fusion cluster

Processor : Intel(R) Xeon(R) CPU E5540 @ 2.53GHz

Number of compute nodes : 320 (Hyperthreading disabled)

Storage : 200 TB of clusterwide disk: 60 TB GFS and 150 TB PVFS

Memory : 304 nodes with 36 GB of RAM, 16 nodes with 96 GB of RAM

OS : Linux kernel 2.6.18-274.18.1.el5

Network : Gigabit ethernet (Intel Corporation 82575EB Gigabit Network Connection)

We can see from Figures 10 and 11 that in the static connection establishment scheme the time taken by `MPI_Init` increases quadratically as the number of processes increases, since $\Theta(N^2)$ connections need to be established for N processes. In the desktop case, the time taken by `MPI_Init` is 25 seconds for 230 processes, and less than a second for up to 60 processes. In the cluster case, `MPI_Init` time is 15 seconds for 230 processes, 282 seconds for 400 processes, and less than a second for up to 100 processes. We can observe that the time taken by `MPI_Init` is more in the desktop case than in the cluster case for the same number of processes. This is due to the following reasons: there are fewer processor cores than the number of processes in the desktop case and more CPU time is required to establish the connections in both the MPICH2 library layer and the network stack layer of the operating system; aspects such as context switch of processes, effect on the cache, page faults during context-switches also likely contribute to the additional cost in the desktop case as the processors are oversubscribed.

On the other hand, the cost of `MPI_Init` of the on-demand connection establishment scheme, however, stays negligible when compared to the static connection scheme. On the cluster, the time taken by `MPI_Init` is between 118 milliseconds and

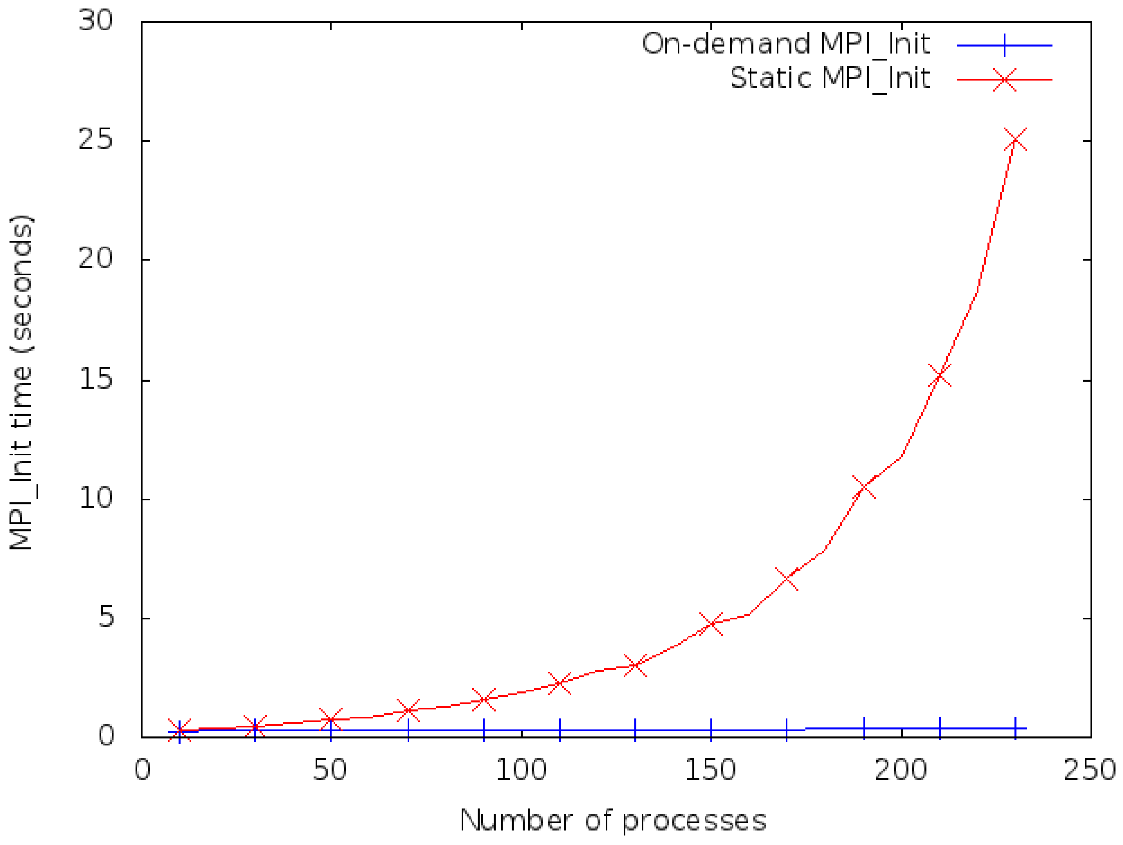

Fig. 10. Cost of `MPI_Init` of the on-demand and static connection schemes in two desktop PCs.

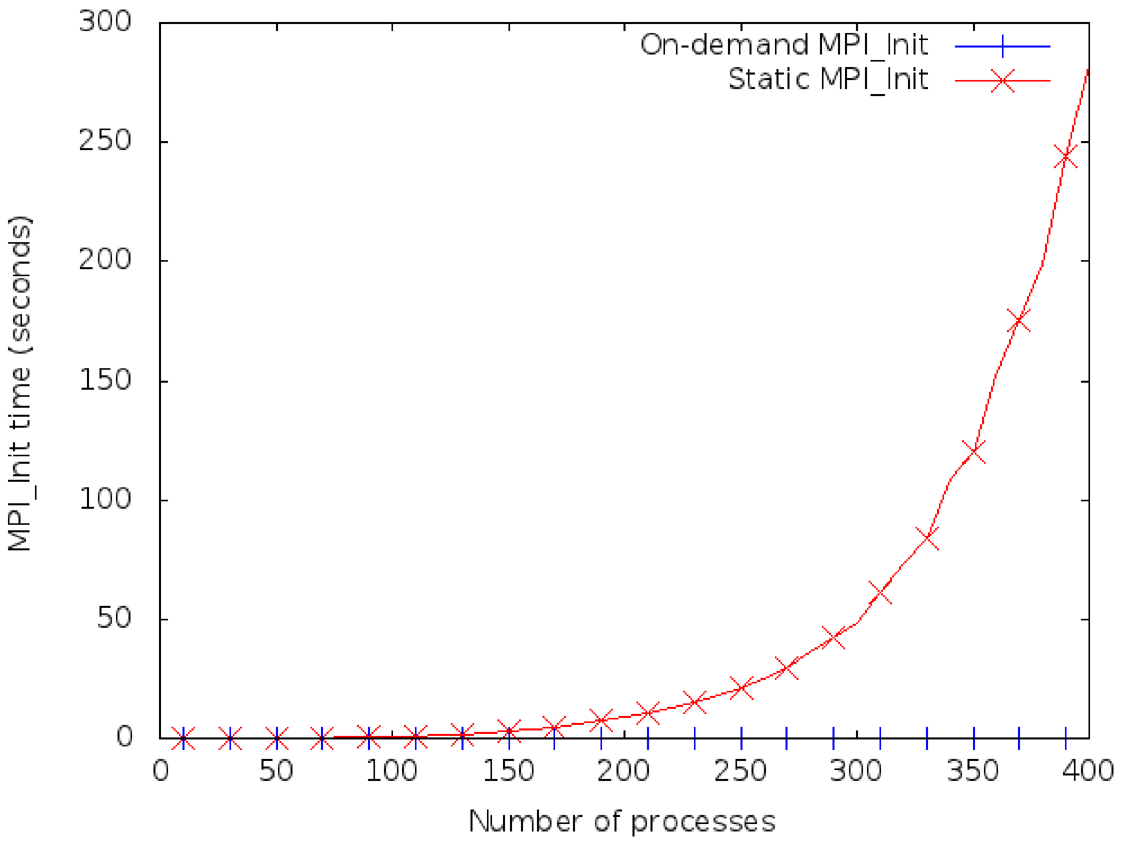

Fig. 11. Cost of `MPI_Init` of the on-demand and static connection schemes in a cluster.

201 milliseconds; the graph of `MPI_Init` time does not follow any pattern as the number of processes increases. The variability in the `MPI_Init` time is mostly due to the operating system overhead. In the desktop case, the `MPI_Init` time varies from 262 to 367 milliseconds, but unlike the cluster case, it follows a linear pattern with a very small slope against the number of processes; this is again due to the oversubscription of processors.

B. Benchmark of Latency in the On-demand Scheme

We showed in the previous section that we reduced the `MPI_Init` time with the on-demand connection scheme. However, not creating connections between all the processes during `MPI_Init` can be expected to cause an increase in the latency of the first communication between any pair of processes, as the connection has to be established between the processes before data exchange can happen. The connection handshake protocol (designed and implemented with a state machine as discussed in Chapter IV) of the on-demand connection scheme can cause some additional latency in the connection establishment time between a pair of processes compared to the static connection scheme, as head-to-head resolutions are likely to happen in the on-demand connection scheme but never in the static connection scheme. Therefore, we created benchmarks to measure this latency.

1. Overhead of Connection Establishment in the On-demand Scheme

A *ping-pong* between two MPI processes is defined as the following: each MPI process sends a message to the other MPI process and then receives a message from the other MPI process. We wrote an MPI program — using the on-demand connection scheme — where two MPI processes do two ping-pongs. The connection establishment between the two MPI processes happen during the first ping-pong; the created connection is used for the second ping-pong. Therefore, the difference between the time taken for the first ping-pong and the second ping-pong is the time taken by the connection establishment handshake protocol. We measured the latency of the first ping-pong and that of the second ping-pong. We ran this program for 10,000 iterations on the two desktop PCs. The average latency of the first ping-pong is 57 microseconds and that of second ping-pong is 42 microseconds; thus, the average latency of the connection establishment protocol is 15 microseconds. The mode latency of the first ping-pong is 19 microseconds, and that of second ping-pong is 3 microseconds; thus, the mode latency of the connection establishment protocol is 16 microseconds.

2. Comparison of the Worst-case Latency of the Static and On-demand Schemes

We wrote another benchmark to measure the worst-case overhead of the connection establishment of the on-demand connection scheme; the worst-case overhead happens in an MPI application that establishes connections between all the processes (i.e., the connection pattern becomes the same as the static connection scheme). In this benchmark program, each MPI process sends a message (using `MPI_Send`) to all the lower-ranked processes and receives a message (using `MPI_Recv`) from all the higher-ranked processes. We took the first time-stamp before `MPI_Init` and the second time-stamp after the `MPI_Send`/`MPI_Recv` sequence. We used `PMI_Barrier` before taking the first time-stamp to ignore the time to launch all the processes. We called `PMI_Barrier` before taking the second time-stamp to ensure that all the processes finish the sends and receives.

This benchmark strategy ensures that the measured latency includes both connection establishment and sends and receives between all the processes — in the same pattern — in both on-demand and static schemes. The only difference is that the connection establishment between all the processes happens during `MPI_Init` in the static scheme whereas it happens during the first ping-pong in the on-demand scheme. We ran this benchmark program five times and computed its average running time. We performed this for different number of processes for both the on-demand scheme and the static scheme. Figure 12 and Figure 13 show the results of this benchmark performed on two desktop PCs and a cluster, respectively.

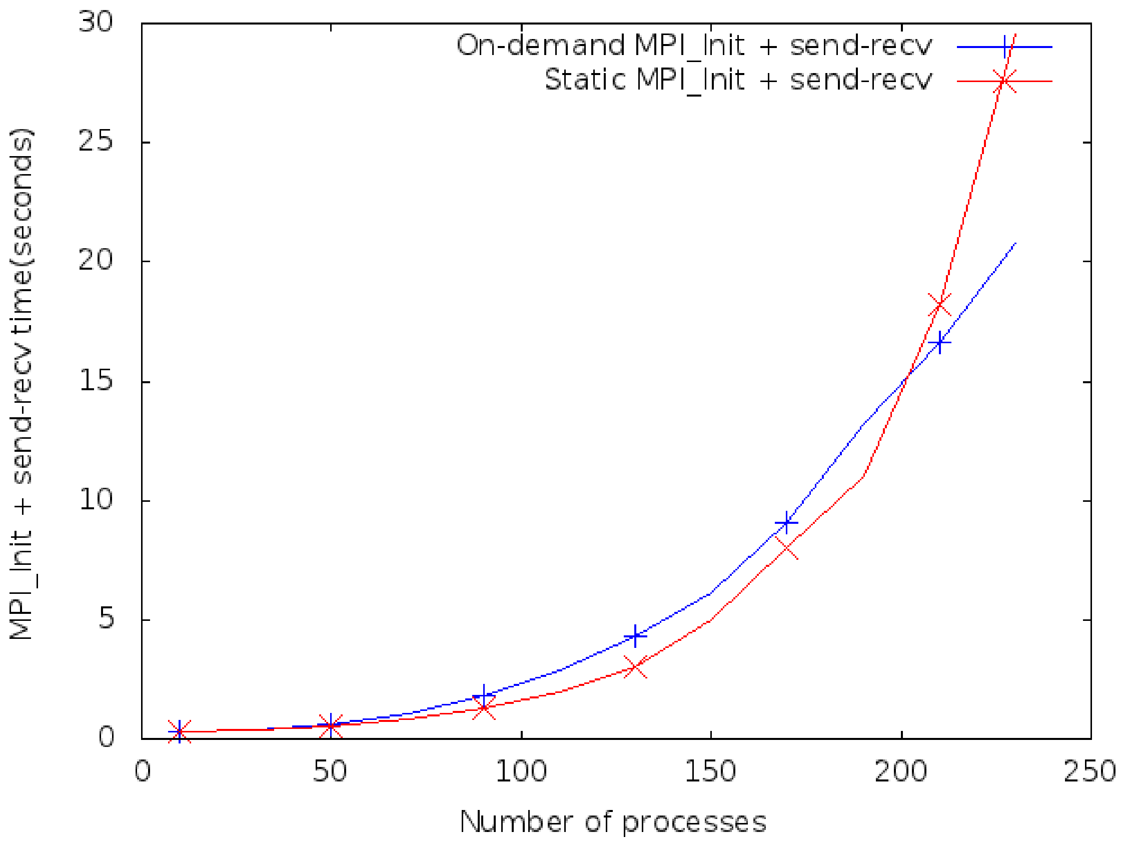

Fig. 12. Comparison of the worst-case (fully-connected) latency of the on-demand scheme with the static scheme on two desktop PCs.

In Figure 12, we observe that the on-demand scheme takes 50 milliseconds more than the static scheme for 50 processes and 2.2 seconds more for 190 processes. However, interestingly, the on-demand scheme takes less time than the static scheme when the number of processes is greater than 190; for instance, for 230 processes, the on-demand scheme takes 8.8 seconds less than the static scheme. In Figure 13, we see that the on-demand scheme takes between 17 milliseconds and 2.5 seconds more than the static scheme when the number of processes varies from 10 to 150 processes. However, the on-demand scheme takes less time than the static scheme when the number of processes is more than 150; for instance, 0.91 seconds less for 170 processes and 31.93 seconds less for 290 processes. In other words, when the latency of the on-demand scheme is more than that of the static scheme, the increase in the latency for the first connection between a pair of processes is not very high and is likely acceptable to most applications. On the other hand, if the number of processes is more than 170 (when the processors are not oversubscribed), the latency of the on-demand scheme is less than that of the static scheme.

As the benchmark program is fully-connected, $N \times (N-1)/2$ connections get created for N processes. We computed the per-connection latency overhead (by dividing the difference in the latencies of the two schemes by the number of connections). In the cluster case, when the on-demand scheme takes longer than the static scheme, the per-connection overhead varies between 39 and 467 microseconds; when the on-demand scheme takes less time than the static scheme, the per-connection gain of the on-demand scheme increases from 304 microseconds to 762 microseconds as the number of processes increases. The per-connection latency overhead is in the order of hundreds of microseconds in the worst-case. Therefore, if the number of destinations per process is small in a parallel application (for example a constant, instead of quadratic, function of the number of processes, as in the worst-case), then the cumulative latency overhead of the first connection between processes in the on-demand

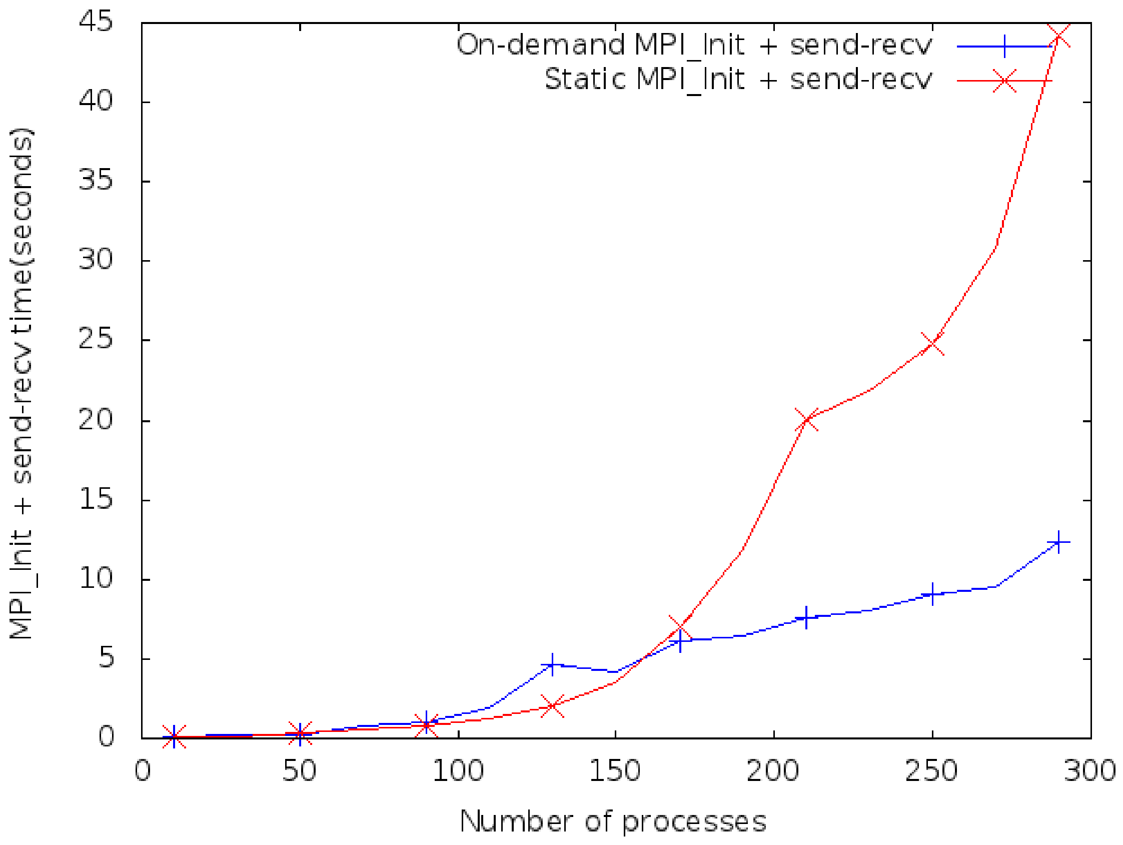

Fig. 13. Comparison of the worst-case (fully-connected) latency of the on-demand scheme with the static scheme on the cluster.

scheme will likely be negligible.

3. Comparison of the Latency of the Static and On-demand Schemes of Typical Parallel Applications

We wrote benchmark applications that simulate the practical large scale parallel applications listed in Table I in Chapter III. In these test applications, each process connects (by issuing `MPI_Send` and `MPI_Recv`) to the same number of destinations as the real-world applications. The latency of `MPI_Init` followed by `MPI_Send` and `MPI_Recv` is measured the same way as explained in Section 2. Our benchmark results, run on the same cluster (for 64 and 400 processes), are shown in Table VIII. We observe that, for 64 processes, the latency of the on-demand scheme is 43% to 75% of the latency of the static scheme. However, for 400 processes, the latency of the on-demand scheme is only 0.07% to 1.83% of the latency of the static scheme. For many of the applications, for 400 processes, the latency for the static scheme is close to five minutes whereas the latency for the on-demand scheme is less than half-a-second.

The latency of the test application simulating "sPPM" for 400 processes with six destinations per process is less than that of the test application simulating "Sphot" for the same number of processes with only one destination because of the variability in the test environment. The other observation is about the latency of the test application simulating "SMG2000" for 400 processes. Even though each process connects to the same number of destinations (i.e., all the other processes) in both the schemes, the latency of the static scheme is 341.06 seconds whereas the latency of the on-demand scheme is only 6.25 seconds. This is because of the following reason: the static scheme uses blocking sockets to `connect` and `accept`, and thus the connection creation process is serialized; the on-demand scheme uses non-blocking sockets and there are multiple instances of the state machines running in each process for different

Table VIII. Comparison of the latency of the on-demand scheme with the static scheme of test applications simulating parallel applications on the cluster.

Test simulating application	Number of processes	Number of of destinations per process	Time taken by static scheme	Time taken by on-demand scheme
sPPM	64	6	0.42	0.21
	400	6	289.48	0.19
SMG2000	64	42	0.34	0.16
	400	399	341.06	6.25
Sphot	64	1	0.31	0.21
	400	1	315.44	0.24
Sweep3D	64	4	0.28	0.21
	400	4	311.04	0.21
Samari4	64	5	0.35	0.15
	400	10	314.77	0.24
CG	64	7	0.30	0.20
	400	11	336.30	0.37

destinations, and all of those state machines make progress in parallel.

As Table I shows, the average number of destinations in most practical large scale MPI applications is typically less than 11, and, in fact, several MPI applications also exhibit the pattern of each process having a constant number of destinations irrespective of the number of processes in the application. Therefore, we observe — from the results of the benchmarks we have discussed in this chapter — that the on-demand connection establishment scheme not only reduces the `MPI_Init` time but also does not affect the latency of communication for typical parallel applications.

CHAPTER VII

CONCLUSIONS

A. Performance Improvement and Scalability of the On-demand Scheme

The typical number of destinations each process has in most MPI applications is very small compared to the total number of processes in the application. Creating all possible connections between all processes during the start-up (`MPI_Init`) unnecessarily increases the start-up times of MPI applications. Connections that are created during the start-up need to be destroyed during shutdown of the application. Besides slower start-up and shutdown, this static connection scheme has further problems as well. In the TCP network module in MPICH2, which was the starting point of our work, unused socket connections in a process imply more use of operating system resources, such as socket descriptors, memory, and kernel buffers of the network stack. This thesis describes a solution to the above problems: a scheme where no connections are created at the start-up time, but instead on-demand as the need for two processes to communicate arises.

We designed and implemented the on-demand connection establishment scheme for the TCP network module of the Nemesis communication subsystem in MPICH2 library. Our benchmarks show that the `MPI_Init` of the on-demand connection scheme takes significantly less time than that of the static connection scheme. The time taken by `MPI_Init` in the static scheme increases quadratically as the number of processes increases; we measured an increase from less than one second for 100 processes to 282 seconds for 400 processes on a high-performance cluster system. On the other hand, the time taken by `MPI_Init` in the on-demand scheme is constant; less than 250 milliseconds for up to 400 processes in the same cluster.

In the on-demand scheme, connections are established between two processes during the first communication (for instance, first `MPI_Send`/`MPI_Recv`). This first communication takes more time than the subsequent communications. We measured this additional latency overhead due to the connection establishment protocol; the average overhead is small (15 microseconds in the cluster).

We also measured the worst-case (where every process connects to every other process) latency overhead of the on-demand scheme in the cluster. We computed the total time taken by `MPI_Init` and an `MPI_Send`/`MPI_Recv` between all the processes in the on-demand scheme and the static scheme. The on-demand scheme takes only up to 2.5 seconds more than the static scheme for up to 150 processes; the on-demand scheme takes up to 4.6 seconds for up to 150 processes. However, the on-demand scheme outperforms (i.e., takes less time than) the static scheme when the number of processes is more than 150 even in a fully-connected application. The per-connection latency overhead computed from this benchmark is in the order of hundreds of microseconds. Therefore, for many practical applications with a small number of destinations per process, the latency overhead of the on-demand scheme is likely negligible.

The on-demand connection scheme is very beneficial for checkpointing/restarting of MPI applications. This feature allows the state of a running process to be stored in a file in storage media and terminating the process and later reconstructing the process with the same state as described in the file. This feature is receiving a lot of attention in the parallel computing community because it improves the fault-tolerance of MPI applications. The support for checkpointing was recently added to MPICH2 as well — the on-demand connection scheme implemented as part of this work is thus very relevant, contributing to a very efficient checkpoint/restart feature.

MPI forum is actively working on the MPI-3 standard to make MPI scalable to million-core systems in the future. As the static scheme is not scalable to hundreds and thousand of nodes today, it will be completely infeasible on a million-core system. The on-demand scheme is needed to make MPI scale to the "monster machines" of the future.

B. Design Verification

The implementation of the on-demand scheme is not trivial. As conventional testing often fails to discover concurrency defects due to a large number of possible thread interleavings, we verified our design formally and thus increased our confidence on the correctness of the system. We modeled (or abstracted) the core state machine of the scheme in PROMELA and verified the model using the SPIN model checker. We verified several *safety properties* (such as no deadlocks, no duplicate connections between two processes) using a few techniques: using *assertions*, SPIN's built-in safety verification mode, and Linear Temporal Logic (LTL) formulae. We also wrote several LTL claims to verify the *liveness properties* that guarantee that a process eventually establishes a connection with another process if it attempts to connect to the other process.

The modeling effort proved useful. We found a few defects thanks to constructing the abstract model of the on-demand connection scheme and SPIN; to correct the defects, both the model and the implementation needed changes. We also applied SPIN's property slicing techniques but did not find any redundancies in the model; property slicing, however, allowed us to replace a few datatypes with smaller ones (in size) to reduce the verification complexity.

From the verification statistics of SPIN, we observe that the number of state transitions varies from one hundred to as high as 194,000 for different LTL claims for different use cases of verification of the model. Even though the state machine did not appear to be exceedingly complex during the design phase, we infer that the complexity of a non-trivial state machine should not be underestimated, and the importance of model-checking should not be neglected.

C. Future Work

Even though we create connections on-demand in our work, the connections are destroyed only when the MPI application is shut down. In a long running application, some connections may become unused and no longer needed. An interesting topic of investigation would be strategies for, and possible benefits of, tearing down unused connections while the application is still running.

Another topic for further investigation is the interplay between checkpointing and on-demand connections. MPICH2's current checkpointing algorithm sends "marker messages" between all the processes of a parallel application. To improve scalability, the messages could be only sent between processes that are connected to each other. The on-demand scheme, however, poses the additional challenge that new connections can arise while the checkpointing algorithm is running.

The aforementioned optimizations are tricky to design, and they would add more states to the state machine. However, the PROMELA model and claims developed in this work may be enhanced and the design can be formally verified before implementing these new features in the library.

We verified only the state machine of the TCP network module in this thesis. The virtual connection has its own state machine in MPICH2. Since the network connection and the virtual connection are associated with each other, we think it would be valuable to also model the virtual connection state machine in PROMELA and verify it with SPIN. Support for *dynamic processes and connections* has been added to the TCP network module of Nemesis recently; this adds additional complexity to the state machine of the on-demand scheme. Verifying this feature with SPIN by enhancing the model may also be fruitful. While verifying virtual connection state machine and dynamic process support, we may be able to add new safety and liveness properties using the states of both the state machines (network connection and virtual connection) together. We anticipate that this effort will overall further improve the reliability of MPICH2.

REFERENCES

[1] H. Sutter and J. Larus, "The free lunch is over: A fundamental turn toward toward concurrency," *Dr. Dobb's Journal*, vol. 30, no. 3, Mar. 2005.

[2] B. C. Brock, G. D. Carpenter, E. Chiprout, M. E. Dean, P. L. De Backer, E. N. Elnozahy, and et al., "Experience with building a commodity Intel-based ccNUMA system," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 207–227, Mar. 2001.

[3] IBM Journal of Research and Development Staff, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1.2, pp. 199–220, Jan. 2008.

[4] D. Trybus, "Design and analyses of a cluster computer," Ph.D. dissertation, University of Western Ontario, Ont., Canada, 2004.

[5] M. Bertozzi, F. Boselli, G. Conte, and M. Reggiani, "An MPI implementation on the top of the Virtual Interface Architecture," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (6th PVM/MPI'99)*, vol. 1697 of *Lecture Notes in Computer Science (LNCS)*, pp. 199–206. Springer-Verlag, Barcelona, Spain, Sep. 1999.

[6] G. F. Pfister, "An introduction to the InfiniBand architecture," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pp. 617–632. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[7] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network: High-performance clustering technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, 2002.

[8] Message Passing Interface Forum, "MPI: A message passing interface standard," Mar 1994, http://www.mpi-forum.org/docs.

[9] MPI Forum, "Message passing interface forum," Accessed in May 2012, http://www.mpi-forum.org.

[10] Argonne National Laboratory, "MPICH2 library: An implementation of MPI," Accessed in May 2012, http://www.mcs.anl.gov/research/projects/mpich2/index.php.

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, and et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (11th European PVM/MPI'04)*, vol. 3241 of *Lecture Notes in Computer Science*, pp. 97–104. Springer, Budapest, Hungary, Sep. 2004.

[12] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386.

[13] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494, 2006.

[14] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *The International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Winter 2005.

[15] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, Boston, MA, 1st edition, 2004.

[16] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, vol. B, pp. 995–1072. North-Holland, Amsterdam, The Netherlands, 1990.

[17] G. Holzmann, "PROMELA man page on Linear Temporal Logic," Accessed in May 2012, http://spinroot.com/spin/Man/ltl.html.

[18] G. J. Holzmann, "A framework for abstraction," in *The SPIN Model Checker*, pp. 230–238. Addison-Wesley, 1st edition, 2004.

[19] IEEE, "POSIX standard," 2001-2008, http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html.

[20] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol 1:The Sockets Networking API*, Addison-Wesley, Boston, MA, 3rd edition, 2003.

[21] W. Gropp, E. Lusk, and R. Thakur, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, Cambridge, MA, 2nd edition, 1999.

[22] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1st edition, 1999.

[23] S. Pakin, "Myrinet," in *Encyclopedia of Parallel Computing*, pp. 1239–1247. Springer, New York, NY, 2011.

[24] D. Ashton, W. Gropp, R. Thakur, and B. Toonen, "The CH3 design for a simple implementation of ADI-3 for MPICH with a TCP-based implementation," Tech. Rep., 2003, http://phase.hpcc.jp/mirrors/mpi/mpich2/docs/tcpadi3.pdf.

[25] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem," in *Proceedings of the Sixth Annual IEEE International Symposium on Cluster Computing and the Grid*, Washington, DC, 2006, pp. 521–530.

[26] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and shared-memory evaluation of MPICH2 over the nemesis communication subsystem," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (13th European PVM/MPI'06)*, vol. 4192 of *Lecture Notes in Computer Science*, pp. 86–95. Springer, New York, NY, 2006.

[27] J. Wu, J. Liu, P. Wyckoff, and D. Panda, "Impact of on-demand connection management in MPI over VIA," in *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, Chicago, IL, 2002, pp. 152–159.

[28] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2002, p. 96.

[29] R. V. D. Geijn, D. Payne, L. Shuler, and J. Watts, "A streetguide to collective communication and its application," Jan. 1996, http://www.cs.utexas.edu/users/rvdg/pubs/streetguide.ps.

[30] RTI International, "The economic impacts of inadequate infrastructure for software testing," Planning Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, May 2002.

[31] D. Rice, *Geekonomics: The Real Cost of Insecure Software*, Addison-Wesley, Boston, MA, 1st edition, 2007.

[32] W. Baziuk, "BNR/NORTEL: Path to improve product quality, reliability and customer satisfaction," in *Sixth International Symposium on Software Reliability Engineering*, 1995, pp. 995–1072.

[33] B. W. Boehm, *Software Engineering Economics*, Prentice Hall, New York, NY, 1st edition, 1981.

[34] M. Robins, "AbortRetryFail AbortRetryFail AbortRetryFail," *New Scientist*, vol. 2265, no. 1, pp. 41–43, 2000.

[35] C. Jones, *Programming Productivity*, Mcgraw-Hill College, New York, NY, 1st edition, 1986.

[36] M. Jones, "What really happened on Mars," Dec. 1997, http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html.

[37] C. Lundstedt, "The large Hadron collider," *Linux Journal*, vol. 2010, no. 199, Nov. 2010.

[38] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[39] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag, New York, NY, 1st edition, 1991.

[40] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*, pp. 1–26. Springer-Verlag, Berlin, Germany, 1st edition, 2008.

[41] E. M. Clarke and E. A. Emerson, "Synthesis of synchronization skeletons for branching time temporal logic," in *Logics of Programs: Workshop*, vol. 131 of *Lecture Notes in Computer Science*. Springer-Verlag, Yorktown Heights, NY, May 1981.

[42] M. Ben-Ari, "A primer on model checking," *ACM Inroads*, vol. 1, no. 1, pp. 40–47, Mar. 2010.

[43] D. Goldrei, *Propositional and Predicate Calculus: A Model of Argument*, Springer, New York, NY, 1st edition, 2005.

[44] M. Fisher, *An Introduction to Practical Formal Methods Using Temporal Logic*, Wiley, Hoboken, NJ, 1st edition, 2011.

[45] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[46] R. Blecksmith, "Implication," Accessed in May 2012, http://www.math.niu.edu/~richard/Math101/implies.pdf.

APPENDIX A

PROMELA MODEL OF THE NEMESIS TCP MODULE STATE MACHINE

```
/*
   Author : Sankara Subbiah Muthukrishnan
   Texas A&M University

   PROMELA model of the TCP network module of the Nemesis
   communication subsytem in MPICH2

  Coding conventions:
  * Use tabs to indent and not spaces.
  * Since "::" is an indentation trigger, indent with a tab
    after "::".
    This keeps the code look neat and remebering to indent is
    easy.
    The main difference between C and PROMELA syntax with
    respect to indentation is that C does not have a
    2-character syntax construct which is an indentation
    trigger but PROMELA does.
  * Use a tabstop=2 for optimal viewing.
  * For all the different verification tests that need to be
    done, use the preprocessor directive in the form "TEST_"
  * Do not use negation or logical and on the preprocessor
    directives.
  * Do NOT nest the preprocessor directives (#if, #elif);use
    them flat
  * The above 3 conventions SHOULD be followed, since the
    automated shell script that verifies this model
    (verify_sm.sh)
    depends on these assumptions. Yes, the shell script does
    some basic parsing to verify all the claims automatically
    and it is not a full-blown pre-processor of PROMELA
*/

#define BUG_FIX 1

#define _empty(_ch)  (len(_ch) == 0)
#define _nempty(_ch) (len(_ch) != 0)
/*
These two macros are defined, since "else" cannot be combined
```

```
with the built-in boolean functions empty and nempty in an
"if" statement.  SPIN throws the following syntax error when
"else" is combined with the builtin "empty" and "nempty":
dubious use of 'else' combined with i/o, saw 'token: ::'
On the other hand, SPIN allows "else" to be used on "len" and
channel poll (including random poll) operations which is not
fully consistent with disallowing "else" to be combined with
nempty and empty. These macros simplify the model which would
be otherwise slightly more complicated and hence these two
macros are used.
*/

#define tVCmsg mtype
#define tNetMsg mtype
#define tNetModState mtype

tNetMsg = {m_close, m_connect, m_connect_ack, m_rank,
  m_rank_ack, m_rank_nak};

tVCmsg = {m_vc_connect, m_vc_disconnect};

typedef tMsg {
  tNetMsg msg_id;
  byte rank;
};

tNetModState = {TS_CLOSED, TC_C_CNTING, TC_C_CNTD,
  TC_C_RANKSENT, TC_C_RANKRCVD, TS_COMMRDY, TS_D_QUIESCENT,
  TA_C_CNTD, TA_C_RANKRCVD};

chan vc_chan[2] = [1] of {tVCmsg};

tNetModState con_state[2] = TS_CLOSED, acpt_state[2] =
  TS_CLOSED;

/*
Note: The variables that are declared globally are done so so
that they can be used in verification claims (LTL/never).
Yes, claims cannot use variables defined in the scope of
proctype or inline functions

Arrays cannot be passed to proctype as an argument, even
within a typedef. That's why a pair of channels cannot be
grouped as a socket (typedef'ed) but they are defined
independently as 2 channels
```

```
atomic or d_step are not used(or needed) in the model, since
the process is single-threaded and it goes through the state
machine from a single loop.
*/

inline send_msg_trans(ch, msg, sm, new_state)
{
  assert(nfull(ch));
  ch!msg;
  sm = new_state;
}

proctype NetModSM(byte id; chan con_snd, con_rcv, acpt_rcv,
  acpt_snd)
{
  tMsg con_msg, acpt_msg;
  byte remote_rank;

end_state:
  do
  ::  ((con_state[id] == TS_CLOSED) &&
      (vc_chan[id]?[m_vc_connect])) ->
        vc_chan[id]?m_vc_connect;
        if
        ::  con_state[id] == TS_COMMRDY ||
            acpt_state[id] == TS_COMMRDY ->
              skip;
        ::  else ->
              con_msg.msg_id = m_connect;
              send_msg_trans(con_snd, con_msg, con_state[id],
                TC_C_CNTING);
        fi

  ::  ((con_state[id] == TS_CLOSED) &&
      (vc_chan[id]?[m_vc_disconnect])) ->
        vc_chan[id]?m_vc_disconnect;

  ::  ((con_state[id] == TS_CLOSED) && nempty(con_rcv)) ->
        con_rcv?_,_;

  ::  con_state[id] == TC_C_CNTING && nempty(con_rcv) ->
        con_rcv?con_msg;
        if
        ::  con_msg.msg_id == m_close ->
```

```
            con_state[id] = TS_D_QUIESCENT;
      ::  con_msg.msg_id == m_connect_ack ->
            con_state[id] = TC_C_CNTD;
      fi;

::  con_state[id] == TC_C_CNTD ->
      if
      ::  nempty(con_rcv) && con_rcv?[m_close, _] ->
            con_rcv?con_msg;
            assert(con_msg.msg_id == m_close);
            con_state[id] = TS_D_QUIESCENT;
      ::  if
          ::  acpt_state[id] == TS_COMMRDY ||
              acpt_state[id] == TA_C_RANKRCVD ->
                con_msg.msg_id = m_close;
                send_msg_trans(con_snd, con_msg,
                  con_state[id], TS_D_QUIESCENT);
          ::  else ->
                con_msg.msg_id = m_rank; con_msg.rank = id;
                send_msg_trans(con_snd, con_msg,
                  con_state[id], TC_C_RANKSENT);
          fi
      fi

::  con_state[id] == TC_C_RANKSENT && nempty(con_rcv) ->
      con_rcv?con_msg;
      if
      ::  con_msg.msg_id == m_close ->
            con_state[id] = TS_D_QUIESCENT;
      ::  con_msg.msg_id == m_rank_ack ->
            con_state[id] = TS_COMMRDY;
      ::  con_msg.msg_id == m_rank_nak ->
            con_state[id] = TS_D_QUIESCENT;
      fi

::  (con_state[id] == TS_COMMRDY &&
    (vc_chan[id]?[m_vc_disconnect] || nempty(con_rcv))) ->
      assert(acpt_state[id] != TS_COMMRDY);
      if
      ::  vc_chan[id]?[m_vc_disconnect] ->
            vc_chan[id]?m_vc_disconnect ->
              con_msg.msg_id = m_close;
              send_msg_trans(con_snd, con_msg,
                con_state[id], TS_D_QUIESCENT);
              acpt_msg.msg_id = m_close;
              send_msg_trans(acpt_snd, acpt_msg,
                acpt_state[id], TS_D_QUIESCENT);
      ::  nempty(con_rcv) ->
            con_rcv?con_msg;
            if
            ::  con_msg.msg_id == m_close ->
                  con_state[id] = TS_D_QUIESCENT;
            fi;
      fi;

::  con_state[id] == TS_D_QUIESCENT ->
      if
      ::  _nempty(con_rcv) ->
            con_rcv?_,_;
      ::  else ->
            con_state[id] = TS_CLOSED;
      fi

::  acpt_state[id] == TS_CLOSED && nempty(acpt_rcv) ->
      acpt_rcv?acpt_msg;
      if
      ::  acpt_msg.msg_id == m_connect ->
            acpt_msg.msg_id = m_connect_ack;
            send_msg_trans(acpt_snd, acpt_msg,
              acpt_state[id], TA_C_CNTD);
      ::  else -> skip;
      fi

::  (acpt_state[id] == TA_C_CNTD &&
    (nempty(acpt_rcv) || con_state[id] == TS_COMMRDY)) ->
      if
      ::  nempty(acpt_rcv) ->
            acpt_rcv?acpt_msg;
            if
            ::  acpt_msg.msg_id == m_close ->
                  acpt_state[id] = TS_D_QUIESCENT;
            ::  acpt_msg.msg_id == m_rank ->
                  acpt_state[id] = TA_C_RANKRCVD;
                  remote_rank = acpt_msg.rank;
            fi
      ::  con_state[id] == TS_COMMRDY ->
            acpt_msg.msg_id = m_close;
            send_msg_trans(acpt_snd, acpt_msg,
              acpt_state[id], TS_D_QUIESCENT);
      fi
```

```
  ::  acpt_state[id] == TA_C_RANKRCVD ->
        if
        :: nempty(acpt_rcv) ->
            acpt_rcv?acpt_msg;
            if
            ::  acpt_msg.msg_id == m_close ->
                  acpt_state[id] = TS_D_QUIESCENT;
            fi
        ::  if
#if BUG_FIX
            ::  con_state[id] == TS_COMMRDY ->
                  acpt_msg.msg_id = m_rank_nak;
                  send_msg_trans(acpt_snd, acpt_msg,
                    acpt_state[id], TS_D_QUIESCENT);
#endif
            ::  con_state[id] == TC_C_RANKSENT ->
                  if
                  ::  id > remote_rank ->
                        acpt_msg.msg_id = m_rank_ack;
                        send_msg_trans(acpt_snd, acpt_msg,
                          acpt_state[id], TS_COMMRDY);
                  ::  else ->
                        acpt_msg.msg_id = m_rank_nak;
                        send_msg_trans(acpt_snd, acpt_msg,
                          acpt_state[id], TS_D_QUIESCENT);
                  fi
            ::  else ->
                  acpt_msg.msg_id = m_rank_ack;
                  send_msg_trans(acpt_snd, acpt_msg,
                    acpt_state[id], TS_COMMRDY);
            fi;
        fi

  ::  (acpt_state[id] == TS_COMMRDY &&
      (vc_chan[id]?[m_vc_disconnect] || nempty(acpt_rcv))) ->
        assert(con_state[id] != TS_COMMRDY);
        if
        ::  vc_chan[id]?[m_vc_disconnect] ->
              vc_chan[id]?m_vc_disconnect ->
                acpt_msg.msg_id = m_close;
                send_msg_trans(acpt_snd, acpt_msg,
                  acpt_state[id], TS_D_QUIESCENT);
                con_msg.msg_id = m_close;
                send_msg_trans(con_snd, con_msg,
                  con_state[id], TS_D_QUIESCENT);
        ::  nempty(acpt_rcv) ->
              acpt_rcv?acpt_msg;
              if
              ::  acpt_msg.msg_id == m_close ->
                    acpt_state[id] = TS_D_QUIESCENT;
              fi;
        fi;

  ::  acpt_state[id] == TS_D_QUIESCENT ->
        if
        ::  _nempty(acpt_rcv) ->
              acpt_rcv?_,_;
        ::  else ->
              acpt_state[id] = TS_CLOSED;
        fi
  od
}

#define p (con_state[0] == TS_COMMRDY)
#define q (acpt_state[0] == TS_COMMRDY)
#define r (con_state[1] == TS_COMMRDY)
#define s (acpt_state[1] == TS_COMMRDY)

#define p_ (con_state[0] == TS_CLOSED)
#define q_ (acpt_state[0] == TS_CLOSED)
#define r_ (con_state[1] == TS_CLOSED)
#define s_ (acpt_state[1] == TS_CLOSED)

#define both_comm_rdy ((p && s && r_ && q_) || \
  (r && q && p_ && s_))
#define both_closed (p_ && s_ && r_ && q_)

#define all_chan_empty (empty(sock[0].ch[0]) && \
  empty(sock[0].ch[1]) && \
  empty(sock[1].ch[0]) && empty(sock[1].ch[1]) && \
  empty(vc_chan[0]) && empty(vc_chan[1]) \
  )

typedef tSocket {
  chan ch[2] = [3] of {tMsg};
};

tSocket sock[2]; /* global shown while replaying; useful */
```

```
init
{
  byte proc_id = 0;

  run NetModSM(proc_id, sock[0].ch[0], sock[0].ch[1],
        sock[1].ch[0], sock[1].ch[1]);
  proc_id++;
  run NetModSM(proc_id, sock[1].ch[0], sock[1].ch[1],
        sock[0].ch[0], sock[0].ch[1]);

#if TEST_1
  vc_chan[0]!m_vc_connect;
#elif TEST_2
  vc_chan[1]!m_vc_connect;
#elif TEST_3
  vc_chan[0]!m_vc_connect;
  vc_chan[1]!m_vc_connect;
#elif TEST_4
  vc_chan[0]!m_vc_connect;
  (p && s && r_ && q_);
  vc_chan[0]!m_vc_disconnect;
#elif TEST_5
  vc_chan[1]!m_vc_connect;
  (r && q && p_ && s_);
  vc_chan[1]!m_vc_disconnect;
#elif TEST_6
  vc_chan[0]!m_vc_connect;
  vc_chan[1]!m_vc_connect;
  (both_comm_rdy && all_chan_empty);
  vc_chan[0]!m_vc_disconnect;
  vc_chan[1]!m_vc_disconnect;
#endif
}

/*
Following is a different way to express the LTL claim p0
(which is same as never claim n0) and verify.

proctype Watchdog()
{
  assert(!(
    ((con_state[0] == TS_COMMRDY) &&
      (acpt_state[0] == TS_COMMRDY)) ||
      ((con_state[1] == TS_COMMRDY) &&
      (acpt_state[1] == TS_COMMRDY))));
}
*/

#if (TEST_1 || TEST_2 || TEST_3 || TEST_4 || TEST_5 || TEST_6)
/*
Same as using Watchdog() process or LTL claim p0a
never n0a {
  do
  ::  ((p && q) || (r && s)) -> break
  ::  else
  od
accept_all:
  skip
}
*/

/* p0a is same as n0a */
ltl p0a  { [] ((p -> !q) && (q -> !p) &&
              (r -> !s) && (s -> !r)) }

/*
p0b is same as p0a, since (q -> !p) is a contra-positive of
(p -> !q) which evaluates to the same in propositional logic.
SPIN's builtin LTL to never converter does not show that
these same. But, the third-party converter ltl2ba shows
that these are same.
*/
ltl p0b { [] ((p -> !q) && (r -> !s)) }
#endif

#if TEST_1
ltl p1a { [] (vc_chan[0]?[m_vc_connect] ->
             <> (p && s && r_ && q_)) }
ltl p1b { [] <> (p && s && r_ && q_) }
#elif TEST_2
ltl p2a { [] (vc_chan[1]?[m_vc_connect] ->
             <> (r && q && p_ && s_)) }
ltl p2b { [] <> (r && q && p_ && s_) }
#elif TEST_3
ltl p3a { [] <>((p && s && r_ && q_) ||
               (r && q && p_ && s_)) }

/* p3b is an equivalence of p3a */
ltl p3b { ([] <> (r && q && p_ && s_)) ||
               ([] <> (p && s && r_ && q_)) }
```

```
#endif

#if (TEST_4 || TEST_5 || TEST_6)
ltl p4a { [] <> both_closed }
#endif
```

VITA

Name:	Sankara Subbiah Muthukrishnan
Address:	Parasol Lab
	Department of Computer Science and Engineering
	Texas A&M University
	College Station, TX 77843-3112
Email Address:	twosquaredfour@gmail.com
Education:	B.E., Computer Science and Engineering (1999)
	College of Engineering, Guindy
	Anna University, Chennai, India
	M.S., Computer Science (2012)
	Texas A&M University, College Station, Texas, USA
Patent Issued:	Usage consciousness in HTTP/HTML for reducing unused data flow across a network
Patent Pending:	Method and apparatus for improving performance and security of DES-CBC encryption algorithm
Experience :	IBM Software Labs (Full-time)
	Argonne National Laboratory (Intern)
	National Instruments (Full-time)

This document was typeset in LaTeX by Sankara Muthukrishnan.