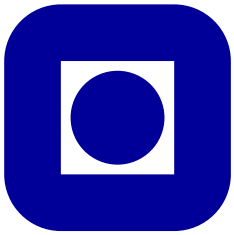
SIF8090P2 Algorithm Construction and Visualization, Specialization

Secure MPI

Jostein Tveit
Jostein.Tveit@idi.ntnu.no

Examine the threats and possible attacks to a cluster and the communication between distant clusters. Outline different methods for encrypted communication of MPI messages, both in a single cluster and between distant clusters communicating over the Internet. With basis in an implementation of MPI, make a proof-of-concept to show the sending and receiving of encrypted MPI messages. Do benchmarks to measure the performance with and without encryption.



Norwegian University of Science and Technology
Dept. of Computer and Information Science

May 13, 2003

Abstract

This paper give an overview of the security threats an MPI cluster can be exposed to. Different attacks and countermeasures are described, and a detailed description of how to implement encryption into the MPICH software package are given. Benchmark results of an encrypted and an unencrypted MPICH cluster are compared and discussed.

Contents

1 Introduction

In the last decade the focus on security in network application has been subject to heavy research, and a lot of new secure software has been developed. The basis of the most common MPI implementation still use insecure network communication and are subject to most of todays well known attacks. Examples on such attacks are eavesdropping and man-in-the-middle attacks. Little effort has been made to make the current MPI implementations secure. Because the MPI implementations are insecure, they must be used in a secure environment with trusted users and a trusted network. If even one evil computer exists on the same network, the network traffic can be eavesdropped and the evil host can cause havoc by inserting malformed packets. The real machines would not even know that the network has been compromised if the attacker take certain precautions.

The idea of securing a standard MPI cluster come from my genuine interest in computer security and the fact that I have been working with clusters the last year. The fact that there are no current work (that I know of) in the area of securing standard MPI implementations also drove my motivation.

This paper is an attempt to describe the possible dangers with an insecure MPI implementation and also give some possible solutions to the problems. The encrypted MPICH solution described, and included in Appendix A, is just a proof-of-concept and is not intended for production environments.

Section 2 give a brief introduction to MPI, current MPI implementations and give the reader an introduction to cryptography and cryptographic methods. Section 3 outline the threats and dangers of an insecure network and try to give solutions to the problems. In Section 4 the methods for securing an MPI cluster are discussed and in particular the modification of the MPICH source code. Section 5 contains benchmark results and a discussion on the solution applied in the implementation section. Section 6 try to give a conclusion about important aspects of MPI and security. Section 7 explain future work in the area of MPI and security and also visions for future implementations of the MPI protocol. Appendix A contains source code used in this project.

2 Background

This section describe and give references to current work in the area of MPI and cryptology. There are not any known implementations of a secure MPI cluster, so this section only describe the basis of the components needed in a secure encrypted MPI environment.

2.1 MPI - Message Passing Interface

MPI is a library specification for message-passing defined by the MPI Forum [MPI]. The MPI standard [For94, Mes98] is used in clusters and parallel machines to be able to execute computer programs i parallel.

Multiple implementations of the MPI standard have been developed and the two most common open source variants are described in Section 2.1.1 and 2.1.2.

2.1.1 MPICH

MPICH [GLDS96, GL96] is an implementation of the MPI standard developed at Argonne National Laboratory as an open source project. MPICH only conform to MPI standard version 1.

The underlying subsystem of MPICH on UNIX machines in an Ethernet MPI cluster is the P4 parallel programming system [BL94, BL92]. The P4 parallel programming system was developed by Ralph M. Butler and Ewing L. Lusk, and is a portable library of C and Fortran subroutines for programming parallel computers. To modify the source code of MPICH to use encrypted communication the P4 subsystem must be altered.

2.1.2 LAM/MPI

LAM/MPI [LAM] is another open source implementation of the MPI standard. It conforms to the MPI-1 and most of the MPI-2 standard, as well as IMPI as described in Section 2.2.

2.2 IMPI - Interoperable MPI

IMPI [IMP00, IMP] is a standard that make different implementations of the MPI standard able to communicate. If the implementations conform to IMPI, they will be able to cooperate and act as a single cluster even though they internally behave differently.

IMPI can also be used in a different scenario. You can build a proxy for connecting two distant clusters and then implement encryption in the proxy. The local traffic inside each cluster will be unencrypted, but the communication between the two clusters become secure. The IMPI standard also ensure that the two clusters can be two different implementations of the MPI standard.

2.3 Cryptography

To secure the network traffic the information sent have to be encrypted. Many different symmetric encryption algorithms exists, but the most widely used are DES [Nat99] and AES [Nat01a]. AES is supposed to replace DES, so the recommended encryption algorithm is AES.

2.3.1 AES - Advanced Encryption Standard

The AES Proposal Rijndael [DR99] was in 2000 chosen by the National Institute of Standard and Technology (NIST) [NIS] to be the new Federal Information Processing Standard, replacing DES and 3DES. The proposal was submitted by Joan Daemen and Vincent Rijmen.

AES is a symmetric block cipher algorithm which can operate in electronic code book (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB) and counter (CTR) mode. For detailed information about the different modes see [Nat01b]. The standard supports data blocks of 128 bits and key sizes of 128, 192 and 256 bits length.

2.4 Key exchange

A good encryption algorithm is not enough. The computers in the cluster need some way to securely agree on a session key used for encryption. A good key exchange algorithm is Diffie-Hellman as described in Section 2.4.1. To be sure that a trusted computer in the cluster talk to another trusted computer in the same cluster, the machines also need to have some sort of authentication towards each other. Authentication is done by the use of a digital signature scheme (See Section 2.4.2) which use asymmetric keys. The public keys of all the hosts in the cluster must be exchanged in a secure way before the actual authentication begin. To avoid this exchange a certificate can be used instead, but this requires a trusted third party.

2.4.1 The Diffie-Hellman Protocol

Diffie-Hellman [DH76] is a simple algorithm for generating a secret key. It uses the difficulty of calculating discrete logarithms in a finite field. The algorithm can not be uses as a encryption/decryption algorithm. The algorithm works as explained in [Sch96]:

Alice and Bob first agrees on a large prime, n and g, such that g is primitive mod n. These integers can be transfered over an insecure channel, and do not have to be secret. Then,

1. Alice chooses a random large integer x and sends Bob: $X = g^x \bmod n$.
2. Bob chooses a random large integer y and sends Alice: $Y = g^y \bmod n$.
3. Alice computes: $k = Y^x \bmod n$.
4. Bob computes: $k' = X^y \bmod n$.

Both k and k' are equal to $g^{xy} \bmod n$, and that value can not be computed by an eavesdropper.

Diffie-Hellman can easily be extended to allow key-exchange between more than two parties (See [Sch96]). This could be extremely useful in a cluster with many nodes which all should agree on the same session key.

There is one problem with this key exchange. There is no way of knowing who you have exchanged the key with. It could easily be an evil man-in-the-middle (See Section 3.2) you have exchanged the key with, and not the real originator. Using authentication, by the means of a digital signature, the problem is solved. A frequently used signature algorithm is described in the next section.

2.4.2 DSS - Digital Signature Standard

Digital Signature Algorithm (DSA) is one of three algorithms described in the Digital Signature Standard [Nat00]. It is based on an asymmetric cipher, and you have a private and a public key.

The message first get hashed by the Secure Hash Algorithm-1 (SHA-1) [Nat02] to create a message digest. The message digest then get signed with the private key, and sent along with the message to the receiver. The receiver can then verify the message by using the signers public key, and compare the message digests (See Figure 1).

The mathematical details of the DSS algorithm can be found in [Nat00].

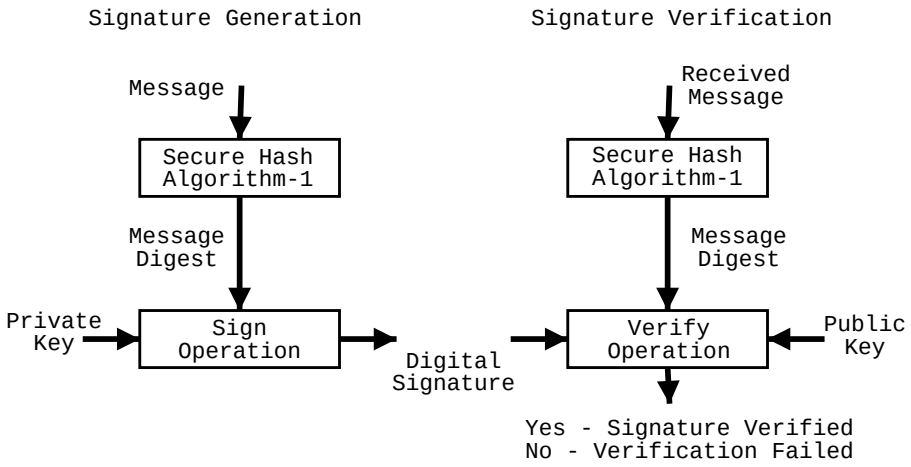


Figure 1: Digital signature generation and verification

2.4.3 Station-to-Station protocol

By combining Diffie-Hellman key-exchange and a digital signature scheme such as DSS, you get a station-to-station protocol as described in [DvOW92]. The protocol assumes that both parties have a signed certificate with the other party's public key. The certificate must have been signed by a trusted third party. The protocol goes as described in [Sch96]:

1. Alice generate a random number, x, and sends it to Bob.

2. Bob generates a random number, y. Using the Diffie-Hellman protocol he computes their shared key based on x and y: k. He signs x and y, encrypts the signature using k. He then sends that, along with y, to Alice: $y, E_k(S_B(x, y))$.

3. Alice also computes k. She decrypts the rest of Bob's message and verifies his signature. Then she sends Bob a signed message consisting of x and y, encrypted in their shared key: $E_k(S_A(x, y))$.

4. Bob decrypts the message and verifies Alice's signature.

Now both parties are authenticated and they share a secret symmetric key. The key could be an AES key as described in Section 2.3.1.

2.5 TLS - Transport Layer Security

Transport Layer Security (TLS) [DA99] based on Netscape's Secure Socket Layer (SSL) is a protocol for securing TCP/IP traffic. The protocol prevent eavesdropping, message tampering and message forgery. It was originally designed to encrypt web traffic, but can be used to secure all traffic on a TCP/IP network. The TLS layers are shown in Figure 2 as the TLS handshake and the TLS record.

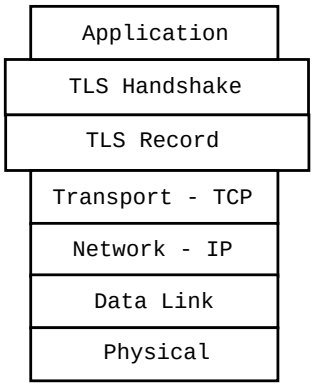


Figure 2: The TLS layer

The TLS record layer provides confidentiality and integrity. It first divides the message into blocks of correct size, then optionally compress the data, before it gets encrypted and sent down to the transport layer. The transport layer must provide a reliable data stream, such as TCP. The TLS record layer can use AES as symmetric encryption algorithm.

The TLS handshake is responsible for authentication, key exchange and negotiation of encryption algorithm and other parameters. It can use Diffie-Hellman as key exchange algorithm and DSA for authentication. Certificates are used to authenticate the machines to each other and the handshake process proceeds as shown in Figure 3. Even though the TLS protocol uses certificates, it is possible to use unsigned public keys, but then the public keys must be securely distributed in other ways.

2.5.1 OpenSSL

OpenSSL [OPE] is an open source implementation of the TLS protocol. It is a full-strength general purpose cryptographic library, and include many different symmetric and asymmetric encryption algorithms, hash functions and support for different types of certificates. Among the supported algorithms are AES, DSA and Diffie-Hellman.

Other security toolkits include Peter Gutmann's cryptlib [CRY] and Mozilla's Network Security Services [NSS].

3 Possible threats and attacks to the MPI protocol

Insecure MPI implementations designed for a TCP/IP network-based cluster are all subject to the standard network attacks, such as packet sniffing, man-in-the-middle attack, replay attack, packet injection and denial-of-service attack. The next subsections describe each of these attacks and give potential countermeasures which can help secure the implementation.

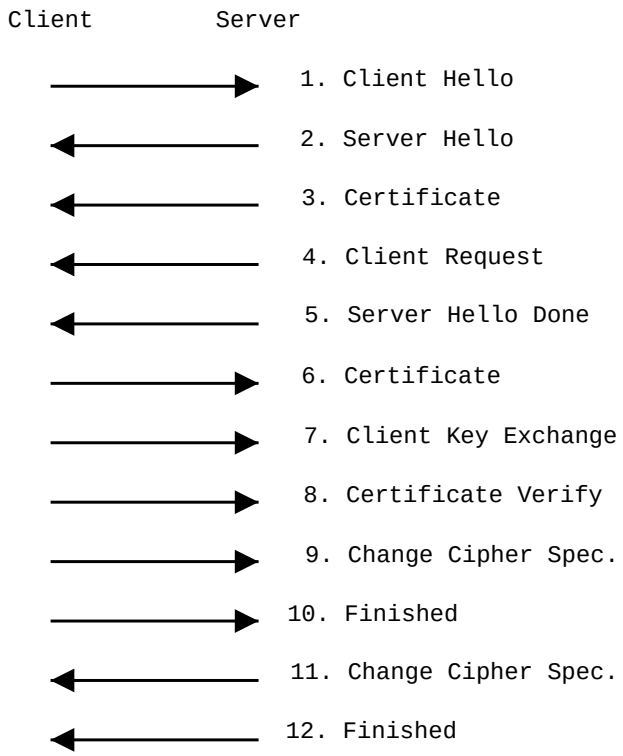


Figure 3: The TLS handshake with mutual authentication

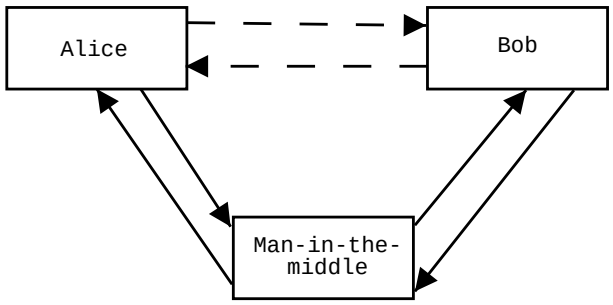


Figure 4: Man-in-the-middle attack

3.1 Packet sniffing

The most common and undetectable attack to a network is packet sniffing. An attacker with access to a computer on a network connected with a switch or a hub, can easily listen to all the traffic on the network. Switched networks are a little harder than networks connected with hubs, but with ARP spoofing [Bel89] in combination with a man-in-the-middle attack, a switched network can be sniffed as well. The easy solution to packet sniffing is to be sure everything important going over the network is encrypted with strong encryption.

3.2 Man-in-the-middle attack

In a man-in-the-middle (MITM) attack (See Figure 4) the attacker places himself in the middle pretending to be the real originator for all the network packets. The attacker can read, alter, duplicate and do whatever he wishes with all the packets when he act as a MITM. Both the server and the client will see all packets as coming from the other party, while the packets really go by the way of the attacker, which can forward the packet to the real originator. Because of this, neither the server nor the client can discover this if they do not take countermeasures.

MITM attacks can be prevented by proper authentication. The machines must either have each others public keys, which must have been transfered by other means than network communication, or the machines can rely on certificates signed by a trusted third party.

3.3 Replay attack

A replay attack is done by copying a network packet and then send the packet to the intended receiver at a later time. The attacker can then trick the computers in the cluster to do some operations more than once. Many operations need to be executed just once, and can cause great disaster if executed more times. The solution to replay attacks is adding timestamps or one-time unique message numbers to the packets sent over the network.

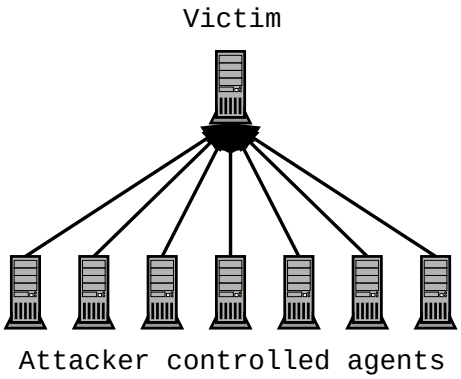


Figure 5: Distributed Denial of Service attack

3.4 Denial of service attack

A denial of service (DOS) attack [SKK+97] is perhaps the most difficult attack to prevent. A DOS attack is performed by connecting to a known service many times. The server then run out of system resources and legitimate connections are denied. An even more dangerous variant is the distributed DOS attack [LRST00] where the attack is coordinated from many different machines at the same time as illustrated in Figure 5.

It is normally difficult to be totally safe against a DOS attack. Because most services actually needs to be offered to the public. Inside a MPI cluster, you usually know the identification of all the nodes, and can then restrict access to the cluster nodes, and block out everybody else. It is still possible to slow down communication if the attacker consume all the available bandwidth between some nodes in the cluster. This is more likely to happen if the nodes are located far away from each other, because you normally do not have control of the interconnecting network, as for example the Internet.

4 Implementation

The MPI protocol itself does not provide any security, and was not designed with security in mind. This does not mean that it is impossible to encrypt MPI traffic. The encryption must either be done by the MPI programmer inside the MPI application or have to be integrated in some way into the MPI implementation. This section outline different ways to integrate security on top of the MPI protocol. The emphasis is on the implementation of encryption inside the MPICH source code, since that was the proof-of-concept implementation.

4.1 Encryption in application

If you need a secure transfer of data in a MPI program, you could of course implement the actual encryption in the MPI application itself. This approach has many disadvantages. It is extremely difficult to get security right, and many programmers will fool themselves by creating programs they think are secure,

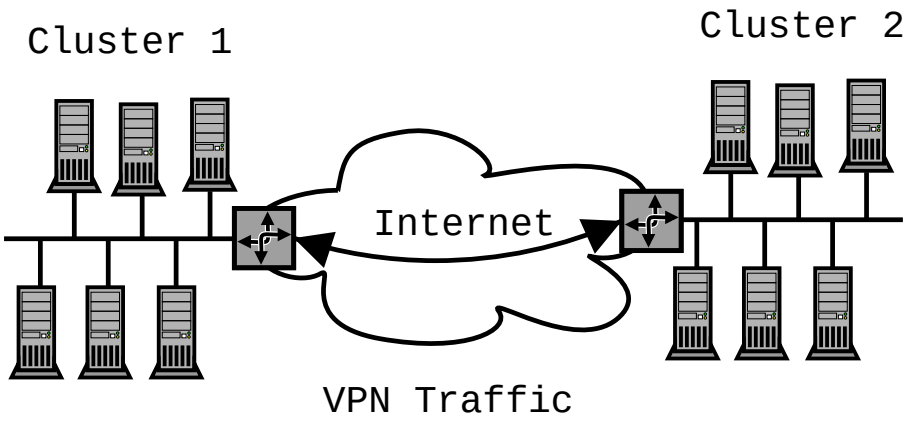


Figure 6: VPN solution between two distant clusters

but actually contains security holes. It it better to know that you have no security at all, than to rely on insecure software. You also have to program everything for each program that require secure communication. It is better to do it once and for all.

One advantage of this approach is that the programmer can decide what to encrypt and what to send as plain-text, and this can probably help on performance. But it is important to be aware that an attacker can get valuable information from data that seems unimportant for everyone else.

4.2 VPN - Virtual Private Network

A Virtual Private Network (VPN) [GLH+00] is a method to provide data integrity and confidentiality between two distant networks or machines. The VPN protocol operate at the network layer and create an encrypted virtual tunnel between the hosts. This solution can be used to forward traffic between two distant clusters over an insecure network by the use of a router or another device which provides VPN. It is also quite fast, as the encryption can be done in specialized hardware.

Inside a single cluster this method can be cumbersome, because you have to set up a VPN connection between every node exchanging information. Because of that, the VPN solution is best used between distant clusters as illustrated in Figure 6. The traffic inside each of the clusters are unencrypted, but the information exchange between the clusters are secure.

Because VPN encrypts traffic at the network layer, every bit of information from the MPI software get encrypted. And there is no danger of information leak from the actual MPI protocol between the two clusters.

4.3 Create a proxy

A more flexible solution than the VPN variant described in Section 4.2, is the use of a proxy to connect two distant clusters. By creating a proxy as outlined in Figure 7 and by the use of the IMPI standard as described in Section 2.2,

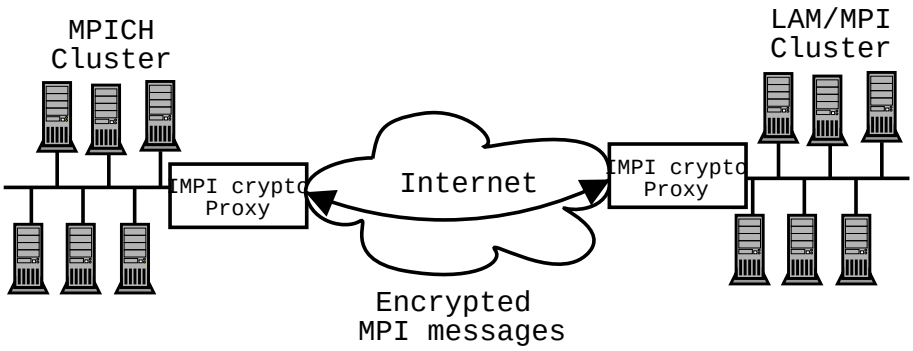


Figure 7: A proxy using IMPI between two distant clusters

it is possible to have two or more different MPI implementations connected as long as they conform to the IMPI standard. Currently LAM/MPI is the only implementation fully compatible with IMPI.

The proxy must be programmed to encrypt and transfer the data transparently among the distant clusters.

This solution has almost the same advantages and drawbacks as the VPN solution. It is difficult to apply to a single cluster, but should work fine between two or more distant clusters, with the advantage of using the IMPI protocol.

4.4 Rewrite an existing implementation

The most user-friendly security approach is perhaps to integrate the security into the MPI implementation. With security embedded in the MPI software, every node in the cluster have the opportunity to encrypt messages. There is no need for specialized hosts for encryption and routing. Everything is out of the box, as long as the authentication and encryption algorithms are agreed on.

Every node in the cluster need to have a private and a public key, which could be generated when installing the software. And every node need to know the public key of every other node in the cluster. The best solution is to use digital signed certificates, and a certification authority which sign all of the nodes public keys. To avoid the use of a certificate, it is possible to blindly trust a public key from a node the first time contact is made, as currently done in the SSH protocol [YKS+02]. If the public key later changes, for example due to a man-in-the-middle attack, the user is warned.

By placing the encryption directly into the software the system administrator can choose to compile the application with or without encryption, according to the users needs. It is also possible to add additional functions to turn the encryption on and off.

The MPICH P4 module, as described in Section 2.1.1, use the network protocol TCP over IP. Because TCP is a reliable data stream it is possible to use the TLS protocol for authentication and encryption. Using a much used and tested open source library as OpenSSL is good, because implementing a cryptographic library is a great challenge and it is almost impossible to do it right and without any security weaknesses at the first try. It is much safer to go with

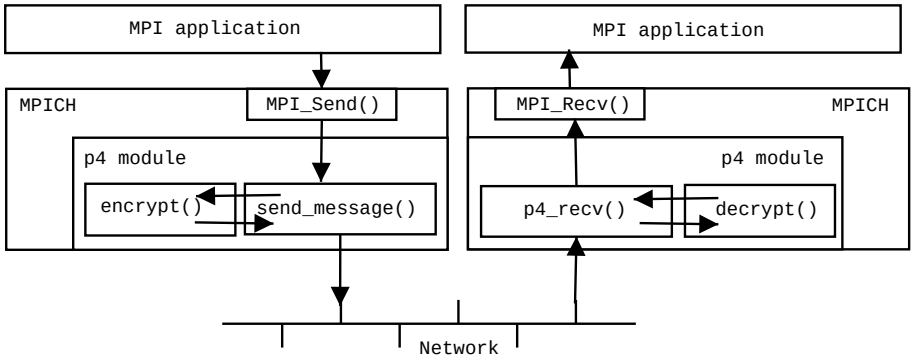


Figure 8: MPICH with P4 subsystem and encryption module

a library which has been thoroughly tested by many users.

To be able to encrypt the MPI traffic between the machines in the cluster the AES algorithm will be used in the proof-of-concept implementation of a secure MPI cluster. An overview of the solution is illustrated in Figure 8.

It is important to think about security at all times. An extremely easy way to compromise the whole security of a cluster is to store private or session keys on a shares NFS disk. NFS use no form for encryption and you could easily send the key in plain-text over the network, and you would not even notice that you did it.

The next sections describe the most important parts of the P4 library, which needs modification in order to include encryption in MPICH.

4.4.1 `p4_msg` struct

The structure `p4_msg` found in `mpid/ch_p4/p4/lib/p4_defs.h` is the basis for a MPI message. The definition of the structure can be seen in Figure 9.

The most important members of the structure are the `len` and `msg` variables. `len` contains the length of the `msg` buffer, while `msg` is the MPI message to send.

By encrypting the content of `msg` before a network send, and decrypt it upon arrival, the MPI message will contain a secure encrypted message. At the same time the `len` variable must be changed to reflect what is actually stored in the `msg` buffer,

4.4.2 `send_message()` function

The function `send_message()` in `mpid/ch_p4/p4/lib/p4_tsr.c` is responsible for sending MPI messages over the network. Because all MPI messages intended for the network goes through this function, this is the place to encrypt the messages. Everything that needs to be done is call the encrypt function as illustrated in Figure 10. A detailed diff of `p4_tsr.c` can be found in Appendix A.1.

```
struct p4_msg {
    struct p4_msg *link;
    int orig_len;
    int type;
    int to;
    int from;
    int ack_req;
    int len;
    int msg_id;
    int data_type;
    int pad;
    char *msg;
};
```

Figure 9: The `p4_msg` structure

```
int send_message(type, from, to, msg, len, data_type, ack_req, p4_buff_ind)
char *msg;
int type, from, to, len, data_type;
P4BOOL ack_req, p4_buff_ind;
{
  ...
  encrypt(msg, len, &tmp_msg, &tmp_len);
  msg = tmp_msg;
  len = tmp_len;
  ...
}
```

Figure 10: Relevant lines from `send_message()` function

```
int
p4_recv(int *req_type, int *req_from, char **msg, int *len_rcvd)
{
  ...
  decrypt(((char *) &(tmsg->msg)),tmsg->len,&tmp_msg,&tmp_len);
  if (tmp_len > tmsg->len)
    printf("Warning: tmp_len > tmsg->len\n");
  for (i=0; i<tmp_len; i++) {
    ((char *) &(tmsg->msg))[i] = tmp_msg[i];
  }
  tmsg->len = tmp_len;
  ...
}
```

Figure 11: Relevant lines from `p4_recv()` function

4.4.3 `p4_recv()` function

The function `p4_recv()` located in `mpid/ch_p4/p4/lib/p4_tsr.c` is responsible for receiving all MPI messages sent over the network. This means that the decryption can be done here. It is important that the decryption is done before any functions try to read from the msg buffer. And the `len` variable must be carefully set to avoid reading in unallocated memory.

Only a few lines need to be added to the `p4_recv()` function. The relevant lines are showed in Figure 11. A detailed diff of the changes done in `p4_tsr.c` can be found in Appendix A.1.

4.4.4 `p4_crypt()` functions

The p4_crypt file is the framework for the P4 crypto functions. The most important functions are the `encrypt()` and `decrypt()` functions enclosed in Appendix A.3. This file is an addition to the MPICH P4 library.

The code inside `encrypt()` and `decrypt()` can be changed according to the desired cryptographic algorithm. For testing purposes the very simple cipher base64 [FB96] was used. The implementation of base64 can be found in Appendix A.5.

The OpenSSL AES implementation was used for cryptographic testing and the OpenSSL functions are called directly from the p4_crypt file. To change the encryption algorithm, you just have to do some minor changes to the file.

4.5 Benchmark program

To measure the performance difference between a MPI program with and without encryption, a small benchmark program was written. This program is included in Appendix A.6. The program just sends and receives a specified number of messages of a specified number of bytes. The root node is responsible for the receiving and all the other nodes sends to the root node.

Test	Number of messages	Message size in bytes	Time in seconds		
			No encryption	Base64	AES ECB (265 bit)
1	1	100000	0.011777	0.035770	0.064087
2	1000	32	1.856256	1.887420	1.914950
3	100	10000	0.098971	0.448579	0.716048

Table 1: Results of benchmark tests

5 Results

To test the implementation the benchmark program described in Section 4.5 was used. The results with different number of messages and size of the messages can be found in table 1.

The test was done on a single computer with three MPI processes. Even though the test was done on a single computer, the send and receive functions use the network interface to connect to the localhost. And since the overhead in encrypting the messages is at the processor and not the network, the results should be quite accurate. It is even possible to compress the data before encryption and thus decrease the network traffic even more. Compressing small messages is probably of no interest.

The three different MPICH configurations used in the benchmark are no encryption, base64 cipher and the AES ECB cryptographic algorithm. No encryption was used to have a reference to compare the other configurations with. It was a standard compiled version of MPICH 1.2.5 that was used. The base64 cipher was used for testing in the implementation and debugging phase of the project, and show the use of an algorithm requiring little computational power. AES in electronic code book (ECB) mode with a 256 bit key was used in the third configuration. The initial Diffie-Hellman key-exchange was omitted due to the time limits of the project. The cost of an initial key-exchange would just add a constant factor to the time measurements, because it would only be done once for all nodes.

The three different tests with different number of messages and different message sizes was executed five times. An average time was computed to prevent small variations from other software running on the same computer.

Test 1 send one single message of 100000 bytes (approximately 98 kB) from two different nodes to the root node. As seen in table 1, the encrypted version is over five times slower than the unencrypted version of MPICH.

Test 2 send 1000 messages of 32 bytes, which is the smallest block size to avoid padding in the OpenSSL AES implementation, from two different nodes to the root node. The time for the encrypted variant is almost the same at the unencrypted, with only three percent difference. This may be because of buffering at some point, either in the network layer or in the MPICH software. When the network receives so many messages at the same time, the overhead in encryption and decryption is much smaller than the buffer queue overhead. There is also overhead in the send and receive process. If a key-exchange algorithm had been implemented, it would probably cause the encrypted variant to take more computational power and thus take slightly longer time.

Test 3 send 100 messages of 10000 bytes (approximately 9.8 kB) from two different nodes to the root node. This test tries to measure a variant between of the two previous tests. The encrypted messages are over seven times slower than the standard version.

6 Conclusions

It is important to know that the security is never stronger than the weakest link. You always have to think of all part of a system in order to get the security correct. And it is not enough to think that encryption solves everything. Issues as physical security and human errors are often a much bigger security risk than the actual network transfer of data. But again, the solution is to think of everything that interact with the system, which can be quite a challenge.

The biggest problem with encryption is that it is time consuming. But you have no real alternatives if you are doing computation on sensitive information. The time spent on encryption is a necessary evil, which can not be avoided.

The benchmarks done in this paper shows a significant speed delay when sending large MPI packets. The encrypted variant of MPICH are five to seven times slower than the unencrypted variant. But an interesting remark is that when sending many small MPI packets the overhead in encryption is minimal. This is probably due to other overheads, either in buffering or setup of the communication.

The overhead with encryption is not normally that important, because you generally have no choice weather to use encryption or not. Sensitive data have to be encrypted, no matter what the overhead is.

Encrypted MPI is probably of more interest in grid-like applications. Grids use open networks and communicate around the world, and thus are more vulnerable to attacks.

The future of MPI, clusters and grid-like applications include security as a great challenge, and security is of no doubt one of the main areas in the years to come.

7 Future Work

To fully secure a cluster a lot of considerations must be taken. This section describe future work and possible extensions to the implemented solution.

A flexible solution require a key-agreement algorithm. The Diffie-Hellman algorithm described i Section 2.4.1 is implemented in the OpenSSL library and could be implemented in the proof-of-concept code. This would make the implementation more useful and not just a proof-of-concept.

An even more secure solution would be to use signed certificates in the key-exchange. This require a trusted third part, which is responsible for the signing of the certificates. Every node must have a built-in public key for this trusted third part. In this way a certificate chain can be made, and the nodes can be sure who they talk to. This kind of infrastructure is called Public Key Infrastructure (PKI) and more info can be found in [AL99]. The PKI solution is probably of no use in a small cluster, but in a big world-wide cluster with many participants, it would be extremely useful to identify the other nodes. PKI is probably the

right choice for the fast growing and popular grid solutions today. An grid-enabled version of MPICH, called MPICH-G2 are discussed in [KTF03]. This MPI implementation use the Globus Toolkit [FK97] for distributed computing.

In current MPI implementations there are no fail-over mode. If one of the nodes crash during execution of a MPI program, the program will terminate, and the program need to be restarted. The MPI software will not try to send the lost data to another node for recomputation. An possible extension, which is almost mandatory for huge world-wide cluster, is to implement a failsafe mode.

Performance with different encryption algorithms could be measured, but this will probably only give slightly different results. Encryption algorithms have already been tested for performance, and the most widely used algorithms are among the fastest available. Different key sizes could give some improvement, but this goes on behalf of the security. Keys smaller than 128 bit should not be used today, because they are vulnerable to brute-force attacks.

More benchmark programs exploring different send and receive algorithms could be tested. This could also test different send and receive structures, as for example broadcast and three-like structures. All of this should be tested in an encrypted environment.

A Source Code

A.1 p4_tsr.diff

```
*** mpid/ch_p4/p4/lib/p4_tsr.c Tue May 13 10:01:53 2003
--- mpid/ch_p4/p4/lib/p4_tsr.c.orig Tue May 13 10:00:26 2003
***************
*** 1,6 ****
  #include "p4.h"
  #include "p4_sys.h"
- #include "p4_crypt.h"

  /*
   * search_p4_queue tries to locate a message of the desired type in the
--- 1,5 ----
*************** int p4_recv( int *req_type, int *req_fro
*** 101,109 ****
  {
      struct p4_msg *tmsg, *tempmsg;
      P4BOOL good;
-     unsigned char *tmp_msg;
-     int tmp_len;
-     int i;

      p4_dprintfl(20, "receiving for type = %d, sender = %d\n",
   *req_type, *req_from);
--- 100,105 ----
*************** int p4_recv( int *req_type, int *req_fro
*** 117,130 ****
   if (!(tmsg = search_p4_queue(*req_type, *req_from, 1)))
   {
       tmsg = recv_message(req_type, req_from);
-
-      decrypt(((char *) &(tmsg->msg)), tmsg->len, &tmp_msg, &tmp_len);
-      if (tmp_len > tmsg->len)
-        printf("Warning: tmp_len > tmsg->len\n");
-      for (i=0; i<tmp_len; i++) {
-        ((char *) &(tmsg->msg))[i] = tmp_msg[i];
-      }
-      tmsg->len = tmp_len;
       /*****
       if (tmsg)
   p4_dprintfl(00, "received type = %d, sender = %d\n",
--- 113,118 ----
*************** P4BOOL ack_req, p4_buff_ind;
*** 445,461 ****
      struct p4_msg *tmsg;
      int conntype;

-     unsigned char *tmp_msg;
```

```
-     int tmp_len;
-
      if (to == 0xffff) /* NCUBE broadcast */
   conntype = CONN_LOCAL;
      else
   conntype = p4_local->conntab[to].type;
-
-     encrypt(msg, len, &tmp_msg, &tmp_len);
-     msg = tmp_msg;
-     len = tmp_len;

      p4_dprintfl(90, "send_message: to = %d, conntype=%d conntype=%s\n",
   to, conntype, print_conn_type(conntype));
--- 433,442 ----
```

A.2 p4_crypt.h

```
#ifndef P4_CRYPT_H
#define P4_CRYPT_H

int encrypt(unsigned char *in_buf,   /* IN  */
    int in_len,              /* IN  */
    unsigned char **out_buf, /* OUT */
    int *out_len             /* OUT */
    );

int decrypt(unsigned char* in_buf,   /* IN  */
    int in_len,              /* IN  */
    unsigned char **out_buf, /* OUT */
    int *out_len             /* OUT */
    );

#endif /* P4_CRYPT_H */
```

A.3 p4_crypt.c

```
#include <stdlib.h>
#include <stdio.h>
#include <openssl/evp.h>

#include "p4_crypt.h"
#include "base64.h"

/* AES key, we would normally set this from a file */
static const unsigned char key[32] =
  {0x12,0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,
   0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,0x12,
   0x56,0x78,0x9a,0xbc,0xde,0xf0,0x12,0x34,
   0x78,0x9a,0xbc,0xde,0xf0,0x12,0x34,0x56};
```

```
int aes_crypt(unsigned char *in,   /* input buffer */
      int inlen,           /* length of input buffer */
      unsigned char **out, /* output buffer */
      int *outlen,         /* length of output buffer */
      int do_encrypt);     /* 1 -> encrypt, 0 -> decrypt */
int base64_encrypt(unsigned char *in_buf, int in_len,
   unsigned char **out_buf, int *out_len);
int base64_decrypt(unsigned char *in_buf, int in_len,
   unsigned char **out_buf, int *out_len);

/*
 * Function to decrypt a message
 */
int decrypt(unsigned char *in_buf, int in_len,
    unsigned char **out_buf, int *out_len)
{
  aes_crypt(in_buf, in_len, out_buf, out_len, 0);
  /* base64_decrypt(in_buf, in_len, out_buf, out_len); */
  return 0;
}

/*
 * Function to encrypt a message
 */
int encrypt(unsigned char *in_buf, int in_len,
    unsigned char **out_buf, int *out_len)
{
  aes_crypt(in_buf, in_len, out_buf, out_len, 1);
  /* base64_encrypt(in_buf, in_len, out_buf, out_len); */
  return 0;
}

/*
 * Function to encrypt/decrypt a message with AES cipher.
 * If do_encrypt is 1, encrypt.
 * If do_encrypt is 0, decrypt.
 */
int aes_crypt(unsigned char *in, int inlen,
      unsigned char **out, int *outlen,
      int do_encrypt)
{
  EVP_CIPHER_CTX ctx;
  int tmplen;

  EVP_CIPHER_CTX_init(&ctx);

  /* Use 256 bits AES Electronic Code Book mode.
   * Does not require initsialization value.
   */
  EVP_CipherInit_ex(&ctx, EVP_aes_256_ecb(), NULL, key,
```

```
    NULL, do_encrypt);

  /* Allow enough space in output for additional block */
  *out = malloc(inlen + EVP_MAX_BLOCK_LENGTH);
  if (*out == NULL) {
    printf("malloc failed\n");
    exit(0);
  }

  if(! EVP_CipherUpdate(&ctx, *out, outlen, in, inlen) ) {
    printf("error Update\n");
    exit(0);;
  }

  /* Buffer passed to EVP_CipherFinal() must be after data just
   * encrypted to avoid overwriting it.
   */
  if(! EVP_CipherFinal_ex(&ctx, *out + *outlen, &tmplen) ) {
    printf("error Final\n");
    exit(0);
  }
  *outlen += tmplen;
  EVP_CIPHER_CTX_cleanup(&ctx);

  return 0;
}


/*
 * Function to "encrypt" a message with base64 encoding.
 */
int base64_encrypt(unsigned char *in_buf, int in_len,
   unsigned char **out_buf, int *out_len)
{
  int status;
  char *buf;
  int len;
  int default_size;

  /* approx size of base64 encoded message */
  default_size = in_len * 1.3;
  buf = malloc(default_size);
  if (buf == NULL) {
    printf("p4_crypt: encrypt malloc failed\n");
    exit(-1);
  }
  while ( (status = encode64(in_buf, in_len, buf,
     default_size, &len)) == -2) {
    /* buffer too small */
    default_size *= 1.3; /* ker med 30% hver gang */
```

```
    buf = realloc(buf, default_size);
    if (buf == NULL) {
      printf("p4_crypt: encrypt realloc failed\n");
      exit(-1);
    }
  }
  if (status != 0) {
    printf("p4_crypt: error in encode64: '%i'\n", status);
    exit(-1);
  }

  *out_buf = realloc(buf, len);
  if (out_buf == NULL) {
    printf("p4_crypt: encrypt realloc failed\n");
    exit(-1);
  }
  *out_len = len;

  return status;
}

/*
 * Function to "decrypt" a message with base64 encoding.
 */
int base64_decrypt(unsigned char *in_buf, int in_len,
   unsigned char **out_buf, int *out_len)
{
  int status;
  char *buf;
  int len;
  int default_size;

  default_size = in_len;
  buf = malloc(default_size);
  if (buf == NULL) {
    printf("p4_crypt: decrypt malloc failed\n");
    exit(-1);
  }
  while ( (status = decode64(in_buf, in_len, buf, &len)) == -2) {
    default_size *= 1.3; /* increase 30% each round */
    buf = realloc(buf, default_size);
    if (buf == NULL) {
      printf("p4_crypt: decrypt realloc failed\n");
      exit(-1);
    }
  }
  if (status != 0) {
    printf("p4_crypt: error in decode64: '%i'\n", status);
    exit(-1);
  }
```

```
  *out_buf = realloc(buf, len);
  if (out_buf == NULL) {
    printf("p4_crypt: decrypt realloc failed\n");
    exit(-1);
  }
  *out_len = len;

  return status;
}
```

A.4 base64.h

```
#ifndef BASE64_H
#define BASE64_H

int encode64 (const char *_in,/* Pointer to original string */
      const int _inlen,/* Length of original string */
      char *_out,/* Pointer to destination buffer */
      int outmax,/* Length of destination buffer */
              int *outlen /* Address of a variable that will
   be modified to take the new
   length of the buffer after e
   ncode is complete */
      );

int decode64 (const char *in,
      const int inlen,
      char *out,
      int *outlen
      );

#endif /* BASE64_H */
```

A.5 base64.c

```
#include "base64.h"

#define OK                  (0)
#define FAIL                (-1)
#define BUFOVER             (-2)
#define CHAR64(c)   (((c) < 0 || (c) > 127) ? -1 : index_64[(c)])
static char basis_64[] =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345678"
  "9+/????????????????????????????????????????????????????????"
  "????????????????????????????????????????????????????????????"
  "??????????????????????";

static char index_64[128] = {
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
```

```
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 62, -1, -1, -1, 63,
  52, 53, 54, 55, 56, 57, 58, 59, 60, 61, -1, -1, -1, -1, -1, -1,
  -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -1, -1, -1, -1, -1,
  -1, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
  41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, -1, -1, -1, -1, -1
};

int encode64(const char *_in, const int _inlen,
     char *_out, int outmax, int *outlen)
{
  const unsigned char *in = (const unsigned char *) _in;
  int inlen = _inlen;
  unsigned char *out = (unsigned char *) _out;
  unsigned char oval;
  char *blah;
  unsigned olen;

  olen = (inlen + 2) / 3 * 4;
  if (outlen)
    *outlen = olen;
  if (outmax < olen)
    return BUFOVER;

  blah = (char *) out;
  while (inlen >= 3) {
    /* user provided max buffer size; don't go over it */
    *out++ = basis_64[in[0] >> 2];
    *out++ = basis_64[((in[0] << 4) & 0x30) | (in[1] >> 4)];
    *out++ = basis_64[((in[1] << 2) & 0x3c) | (in[2] >> 6)];
    *out++ = basis_64[in[2] & 0x3f];
    in += 3;
    inlen -= 3;
  }
  if (inlen > 0) {
    /* user provided max buffer size; don't go over it */
    *out++ = basis_64[in[0] >> 2];
    oval = (in[0] << 4) & 0x30;
    if (inlen > 1)
      oval |= in[1] >> 4;
    *out++ = basis_64[oval];
    *out++ = (inlen < 2) ? '=' : basis_64[(in[1] << 2) & 0x3c];
    *out++ = '=';
  }

  if (olen < outmax)
    *out = '\0';

  return OK;
```

```
}

int decode64(const char *in, const int inlen,
     char *out, int *outlen)
{
  unsigned len = 0, lup;
  int c1, c2, c3, c4;

  if (in[0] == '+' && in[1] == ' ')
    in += 2;

  if (*in == '\0')
    return FAIL;

  for (lup = 0; lup < inlen / 4; lup++) {
    c1 = in[0];
    if (CHAR64 (c1) == -1)
      return FAIL;
    c2 = in[1];
    if (CHAR64 (c2) == -1)
      return FAIL;
    c3 = in[2];
    if (c3 != '=' && CHAR64 (c3) == -1)
      return FAIL;
    c4 = in[3];
    if (c4 != '=' && CHAR64 (c4) == -1)
      return FAIL;
    in += 4;
    *out++ = (CHAR64 (c1) << 2) | (CHAR64 (c2) >> 4);
    len++;
    if (c3 != '=') {
      *out++ = ((CHAR64 (c2) << 4) & 0xf0) | (CHAR64 (c3) >> 2);
      len++;
      if (c4 != '=') {
*out++ = ((CHAR64 (c3) << 6) & 0xc0) | CHAR64 (c4);
len++;
      }
    }
  }

  *out = 0;
  *outlen = len;

  return OK;
}
```

A.6 benchmark.c

```
/*
 * Simple benchmark program to time send/recv of MPI messages.
```

```
 *
 * The program accept the number of MPI messages to send from
 * each node to the root node, and also the size of each message.
 *
 * Usage: benchmark <number of messages to send from each node>
 *                  <message size in bytes>
 *
 * Jostein Tveit <Jostein.Tveit@idi.ntnu.no>
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
  int my_rank;            /* Rank of process */
  int processes;          /* Number of processes */
  int source;             /* Rank of sender */
  int dest;               /* Rank of reciever */
  int tag = 0;            /* Tag for all messages */
  char *message;          /* Pointer to message to send */
  int message_size;       /* Message size */
  int number_of_messages; /* Number of messages */
  MPI_Status status;      /* Return status for recieve */
  int i;
  int count;              /* Count variable */
  double start_time;      /* Variable for timing */
  double end_time;        /* Variable for timing */

  /* Start up MPI */
  MPI_Init(&argc, &argv);
  /* Find out process rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Find out number out processes */
  MPI_Comm_size(MPI_COMM_WORLD, &processes);

  /* Command line argument check */
  if (argc != 3) {
    printf("Usage: %s <number of messages to send from "
   "each node> <message size in bytes>\n",
   argv[0]);
    exit(-1);
  }

  /* Read in command line arguments */
  number_of_messages = atoi(argv[1]);
  message_size = atoi(argv[2]);

  if (my_rank == 0) {
```

```
    /* Allocate memory for message */
    message = malloc(message_size);
    if (message == NULL) {
      printf("malloc error\n");
      exit(-1);
    }

    /* Start timing */
    start_time = MPI_Wtime();

    /* Receive number_of_messages from each process */
    count = 0;
    for (source = 1; source < processes; source++)
      for (i = 0; i < number_of_messages; i++) {
MPI_Recv(message, message_size, MPI_CHAR, source, tag,
 MPI_COMM_WORLD, &status);
count++;
      }

    /* End timing */
    end_time = MPI_Wtime();

    printf("Process 0 recieved a total of %i messages of "
   "%i bytes from %i processes\n"
   "Total time: %f seconds\n",
   count, message_size, processes-1,
   end_time - start_time);

    /* Free memory allocated for message */
    free(message);

  } else { /* my_rank != 0 */

    /* Allocate memory for message */
    message = malloc(message_size);
    if (message == NULL) {
      printf("malloc error\n");
      exit(-1);
    }

    /* Create message */
    for (i = 0; i < message_size; i++)
      message[i] = 'A';

    /* Send message to process 0 */
    dest = 0;
    for (i = 0; i < number_of_messages; i++)
      MPI_Send(message, message_size, MPI_CHAR,
       dest, tag, MPI_COMM_WORLD);
```

```
    /* Free memory allocated for message */
    free(message);
  }

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
}
```

References

[AL99] Carlisle Adams and Steve Lloyd. *Understanding the Public-Key Infrastructures: Concepts, Standards, Deployment Considerations*. New Riders, 1999.

[Bel89] Steven M. Bellovin. Security problems in the tcp/ip protocol suite. *Computer Communications Review*, 19(2):32–48, apr 1989.

[BL92] R. Butler and E. Lusk. User's Guide to the p4 Parallel Programming System. ANL 92/17, Argonne National Laboratory, Illinois, USA, 1992.

[BL94] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Journal of Parallel Computing*, 20(4):547–564, apr 1994.

[CRY] cryptlib webpage. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[DA99] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246, jan 1999.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[DR99] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael*, 1999.

[DvOW92] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[FB96] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045, nov 1996.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[For94] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.

[GL96] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[GLH+00] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. *A Framework for IP Based Virtual Private Networks*. RFC 2764, feb 2000.

[IMP] IMPI webpage. `http://impi.nist.gov/IMPI/`.

[IMP00] IMPI Steering Committee. *IMPI: Interoperable Message-Passing Interface*, 2000.

[KTF03] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. To appear in Journal of Parallel and Distributed Computing (JPDC), 2003.

[LAM] LAM/MPI webpage. `http://www.lam-mpi.org`.

[LRST00] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, pages 2275–2280, oct 2000.

[Mes98] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[MPI] MPI Forum webpage. `http://www.mpi-forum.org`.

[Nat99] National Institute of Standards and Technology (NIST). *Data Encryption Standard (DES) (FIPS PUB 46-3)*, October 1999.

[Nat00] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS) (FIPS PUB 186-2)*, January 2000.

[Nat01a] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard (AES) (FIPS PUB 197)*, November 2001.

[Nat01b] National Institute of Standards and Technology (NIST). *Recommendations for Block Cipher Modes of Operations, Methods and Techniques*, December 2001.

[Nat02] National Institute of Standards and Technology (NIST). *Secure Hash Standard (FIPS PUB 180-2)*, August 2002.

[NIS] National institute of standards and tecnology (NIST) webpage. `http://www.nist.gov/`.

[NSS] Network Security Services (NSS) webpage. `http://www.mozilla.org/projects/security/pki/nss/`.

[OPE] OpenSSL webpage. `http://www.openssl.org`.

[Sch96] Bruce Schneier. *Applied Cryptography (Second Edition)*. John Wiley & Sons, 1996.

[SKK+97] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society, IEEE Computer Society Press, may 1997.

[YKS+02] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. *SSH Protocol Architecture*. IETF Network Working Group, sep 2002. Internet-Draft, work in progress.