Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

TCP Input Threading in High Performance Distributed Systems

Hans H. HAPPE

Department of Mathematics and Computer Science,
University of Southern Denmark, DK-5230 Odense M, Denmark

hhh@imada.sdu.dk

Abstract. TCP is the only widely supported protocol for reliable communication. Therefore, TCP is the obvious choice when developing distributed systems that need to work on a wide range of platforms. Also, for this to work a developer has to use the standard TCP interface provided by a given operating system.

This work explores various ways to use TCP in high performance distributed systems. More precisely, different ways to use the standard Unix TCP API efficiently are explored, but the findings apply to other operating systems as well. The main focus is how various threading models affect TCP input in a process that has to handle both computation and I/O.

The threading models have been evaluated in a cluster of Linux workstations and the results show that a model with one dedicated I/O thread generally is good. It is at most 10% slower than the best model in all tests, while the other models are between 30 to 194% slower in specific tests.

Keywords. Distributed systems, HPC, TCP

Introduction

The Transmission Control Protocol (TCP) has become the de facto standard for reliable Internet communication. As a result, much work has gone into improving TCP at all levels (hardware, kernel, APIs, etc.). This makes TCP the only viable choice for distributed applications that need to be deployed outside the administrative domain of the deployer. I.e., one might have gained access to a remote cluster, but this does not mean that the administrator is willing to meet specific communication requirements (protocols, operating systems). Grid and peer-to-peer are other examples of environments with this nature.

TCP was designed as a reliable connection-oriented client-server protocol. From the users point of view, TCP provides a way to stream bytes between two endpoints. Therefore, the user has to provide a stream encoding mechanism that ensures separation of individual messages (message framing). Encoding and decoding streams make TCP development more complicated and it can lower performance.

Message framing and other issues have been addressed in emerging protocols like the Stream Control Transmission Protocol (SCTP) [1] and the Datagram Congestion Control Protocol (DCCP) [2]. These protocols still need to mature and become generally available. This leaves TCP as the only choice.

This work describes and evaluates various ways to use TCP communication in high performance distributed systems. Particularly in systems where nodes act as both client and server. In this context a node is a Unix user-process that uses the kernel TCP API for communication. This duality raises the question of threading. If a client is I/O-bound, with regards to communication, it can take on the role as server while waiting. This will avoid the

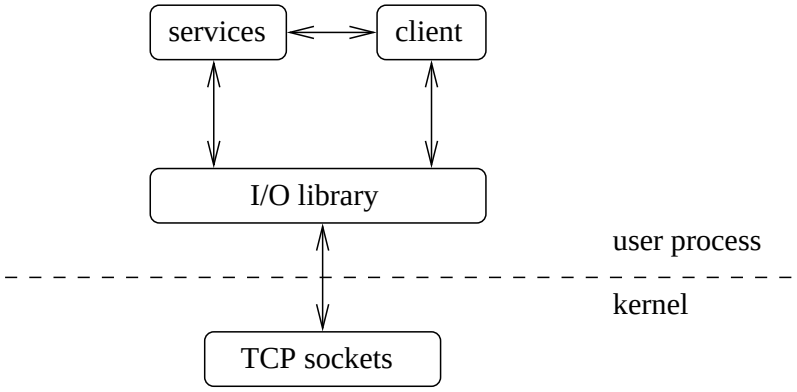

Figure 1. System overview.

overhead of context switching. In other scenarios multiple threads could be a better option. These different threading models are the main focus of this work.

The work generally applies to a wide range of distributed systems that are based on TCP communication. Especially systems where nodes concurrently have to handle tasks outside the communication context, might benefit from this work. High performance message passing [3,4] and software-based distributed shared memory [5,6] systems fit into this category. Grid and peer-to-peer based systems could also benefit from this work.

1. System Overview

Basically a distributed system consists of multiple interacting nodes. Each node is responsible for a subset of the system and might be a client entry point to the system.

In the context of this paper a node is a process in an operating system, which can have multiple threads of execution all sharing its address space. An I/O library will handle TCP communication with other processes.

Figure 1 gives a simple overview of the different components in a process. Services handle the distributed system responsibilities of the process. This includes communication, protocols, storage, etc. In some cases it is convenient that clients can become part of the process. In these cases performance and/or simplicity are more important than client separation.

2. Unix TCP Communication

The basis for TCP communication in Unix is the socket, which is a general abstraction for all types of network related I/O. As most kernel resources, a socket is referenced from user-space by a file descriptor that is valid until the user explicitly closes the socket. Before actual communication can start a TCP socket has to be connected to the other end. Now data can be streamed between the endpoints by reading and writing to the sockets.

2.1. Sending

Sending a message is very simple because the decision to send implies that the content and context of the message is known. Basically the content just needs to be encoded into a message format that can be decoded at the receiver. Then the message can be written to the socket that represents the destination.

Writing to a TCP socket will copy the data to an in-kernel TCP buffer, but in case this buffer is full the connection is saturated. Obviously, this could result in deadlocks if not handled carefully. Avoiding deadlocks in distributed systems is a system design issue that

can not universally be solved by a communication abstraction (I/O library). Features like buffering can help to avoid deadlocks, but in the end it is the system design that should guarantee deadlock-free operation based on these features.

2.2. Receiving

It is a general fact in communication that the receiving side is harder to handle. Initially the receiver is notified about pending input, but only after the input is read can its context be determined. I.e., the kernel needs to process IP and TCP headers in order to route input to the correct destination socket. A similar kind of input processing has to be done in user-space in order to route data to the correct subsystem of an application.

2.3. Monitoring Multiple TCP Sockets

The fact that each TCP connection is represented by one file descriptor, poses the question of how to monitor multiple connections simultaneously.

Having one thread per socket to handle input is a simple way to monitor multiple sockets. This trades thread memory overhead for simplicity. While this memory overhead might be acceptable it can also result in context switching overhead, which in turn pollutes the CPU cache and TLB (multiple stacks). In practice a thread waits for input by doing a blocking read system call on the socket. When the kernel receives data for the socket, it copies it to the buffer provided by the read call and wakes up the thread that now can return to user-space. This ”half” system call (return from kernel) is a short wakeup path and the input data will be available upon return.

Most operating systems provide a way for a single thread to monitor multiple sockets simultaneously. Unix systems generally provide the system calls *[*poll() and *select()*, but these has scalability issues [7]. Therefore, various other scalable and non-standard methods has been invented. Linux provide the *epoll* [8] mechanism which is a general way to wait for events from multiple file descriptors. Basically, a thread can wait for multiple events in a single system call (*epoll_wait()*). The call returns with a list of one or more ready events that need to be handled. In the socket input case an event is handled by doing a read on the ready socket. Compared to the multi threaded model described above this is a whole system call per socket in addition to the *epoll_wait()* system call. This overhead should be smaller than the context switching overhead in the multi threaded model for this single threaded method to be an advantage. With a low input rate or single socket activity this will not be the case.

3. Input Models

Both services and clients can start large computations as a result of new input. If communication should continue asynchronously during these computations, multiple threads are required. The best way to assign threads depends on the specific distributed system.

The focus of this paper is a system where clients has their own thread. The thread might do communication or service work, but only when the client calls into the I/O library or a service. From the client’s point view this is a natural design, because it controls when to interact with the distributed system.

Another characteristic is that services are I/O bound. Basically they function as state machines acting on events from the I/O library and/or the client, without doing much computation. Extra threads could be added to handle service computations, but this will not be addressed in this paper.

Figure 2 illustrates common input cases. Case a) is input directed to a service or a client without producing new output (response/forward). This can be handled in the context of the

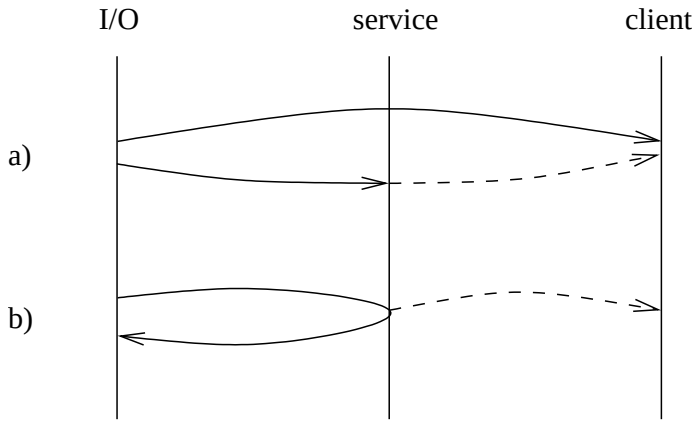

Figure 2. Input scenarios. Dashed arrows indicate events that might follow.

client thread when it is ready, because the input event does not affect other parts of the system. In case b) a service produces new output as a result of the input. This output could be a response to a request or some sort of forwarding and might be important for other nodes. The input should therefore be handled as soon as possible and not have to wait for the client to be available for communication. This requires at least one extra thread for input handling.

The following sections describe the threading models that will be evaluated in section 4. Only the input path is described, while details about sending and setting up connections are left out.

3.1. Model 1: Single Thread

In this model I/O and service processing are only handled when the client thread calls into these. When such a call can not be served locally the client thread will be directed to the I/O library in order to handle new input. When input arrives it will be handled and in case it matches the requirements of the client, control is returned to the client (Figure 3).

Case a) is handled perfectly because context switches are avoided when input for the client arrives. However, in case b) progress in the overall system can be stalled if the client is CPU-bound. Also, this model requires that multiple sockets can be monitored simultaneously as described in section 2.3.

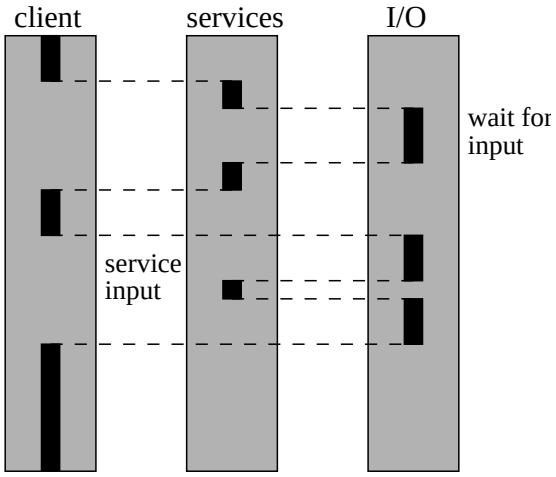

Figure 3. Single thread model.

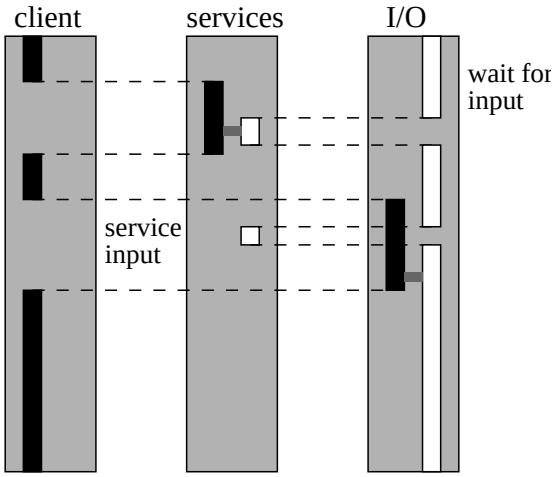

Figure 4. Input thread model. The white thread is the input thread.

3.2. Model 2: Input Thread

In this model a thread is used for input handling. The thread starts in the I/O library and when input arrives it delivers this to one of the services or the client (Figure 4). The exact details of how this multiplexing is done is not important in this context. In case the client is waiting for input the input thread must wake up the client when this input arrives. This adds context switching overhead, but solves the issues with input case b). Again, this model requires that multiple sockets can be monitored simultaneously.

3.3. Model 3: A Thread per Socket

This model works similarly to M2 except that each socket has a dedicated input thread. This removes the overhead of monitoring multiple sockets, but also introduces new issues as described in section 2.3.

3.4. Models 1 and 2: Hybrid

Given the cons and pros of the described models a hybrid between M1 and M2 would be interesting. The idea is to make the client handle I/O events while it is otherwise waiting for input. This requires a way to stop and restart the input thread by request from the client thread. While this is possible, it can not be done in a general way without producing context switches. The problem is that the input thread has to exit and reenter the event monitoring system call. New kernel functionality is needed in order for this model to work and it will therefore not be evaluated in this paper.

4. Evaluation

The evaluation was done with a software-based distributed shared memory system, which currently is work in progress. It is based on the PastSet memory model [6] and fits into the system model described in section 1. The memory subsystem is implemented as services and applications act as clients using these services.

The results show the performance of the various models for this specific distributed system. Performance variations in these results have not been examined in close detail, but some hints to why models perform differently are given in the description. Low-level information about cache misses and context switches would be interesting if the goal was to improve operating systems, but this work targets the use of generally available communication methods.

4.1. Application

A special evaluation application that can simulate different computation and communication loads has been developed. Basically each process runs a number of iterations that have a communication and a computation part. How these parts work in each run are specified by load-time parameters.

In the communication part one process writes some data to shared memory and all others read this data (like multicasting). The writer in each iteration is chosen in a round-robin fashion. A parameter $comm$ defines how much data is written in each iteration.

The computation part does a series of local one byte read/update calculations. A parameter mem defines how much memory is touched by these calculations and is therefore the minimum number of read/updates done in each iteration. This makes it possible to test the effect of memory use in computations. Another parameter $calc$ defines a maximum number of read/updates that will be done, but the actual number of read/updates carried out in each iteration is chosen randomly from the range $[mem; calc]$. This makes computations uneven and ensures that processes are not in sync. A pseudo-random number generator is used to ensure comparability and the generator is initialized with different seeds on for each process.

4.2. Test Platform

The evaluation was performed on a 32 node Linux cluster interconnected by Gigabit Ethernet. Each node had an Intel Pentium 4 541 64-bit CPU with Hyper-Threading and ran version 2.4.21 of the Linux kernel. The kernel supported the new Native POSIX Threading Library (NPTL) [9] and was used in the evaluation.

Hyper-Threading was turned off so that the multi threaded models did not get an advantage. This was done by forcing processes to stay on one CPU with the *taskset(1)* tool.

The less scalable *poll()* system call was used to monitor multiple sockets, because the kernel did not support *epoll* (available in versions 2.6.x). This could have a negative effect on the performance of the M1 and M2 models.

4.3. Results

The tests were done by running a series of communicate/compute iterations (see section 4.1) and hereby measure the average iteration time. Each test were run three times to test for variations between runs. The variations were insignificant and therefore the average of these three runs are used in the following results.

4.3.1. I/O-bound

Figures 5 and 6 show the scalability of the different models without computation. M1 performs better than the other two threaded models, as expected. It is only marginally better than M2 and the difference is not even visible when communication increases (Figure 6). Therefore, M2 only imposes a small context switching overhead compared to M1. The many threads in M3 give even more overhead as the number of nodes increase (Figure 5). With added communication and therefore higher memory utilization the overhead of threading really decreases performance (Figure 6). With 32 nodes M3 almost triples the completion time compared to the other models. These observations indicate that thread memory overhead (stacks and task descriptors) is the cause of M3's performance issue.

When plotting time as a function of communication load, the poor performance of M3 becomes even more apparent (Figure 7). M1 and M2 do equally well, while there is an anomaly starting at 16KB. M3 does not have this anomaly which indicates that the monitoring of multiple sockets (*poll()*) is the cause. Longer I/O-burst increases the chance that there

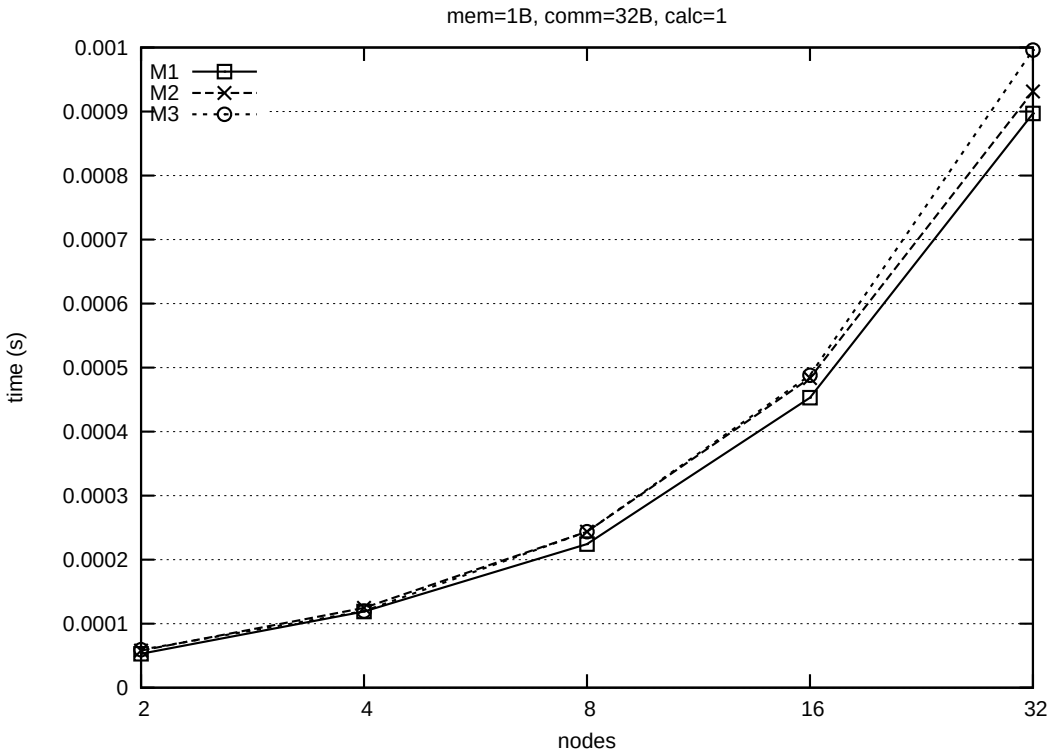

Figure 5. Scalability with 32B read/write and no computation.

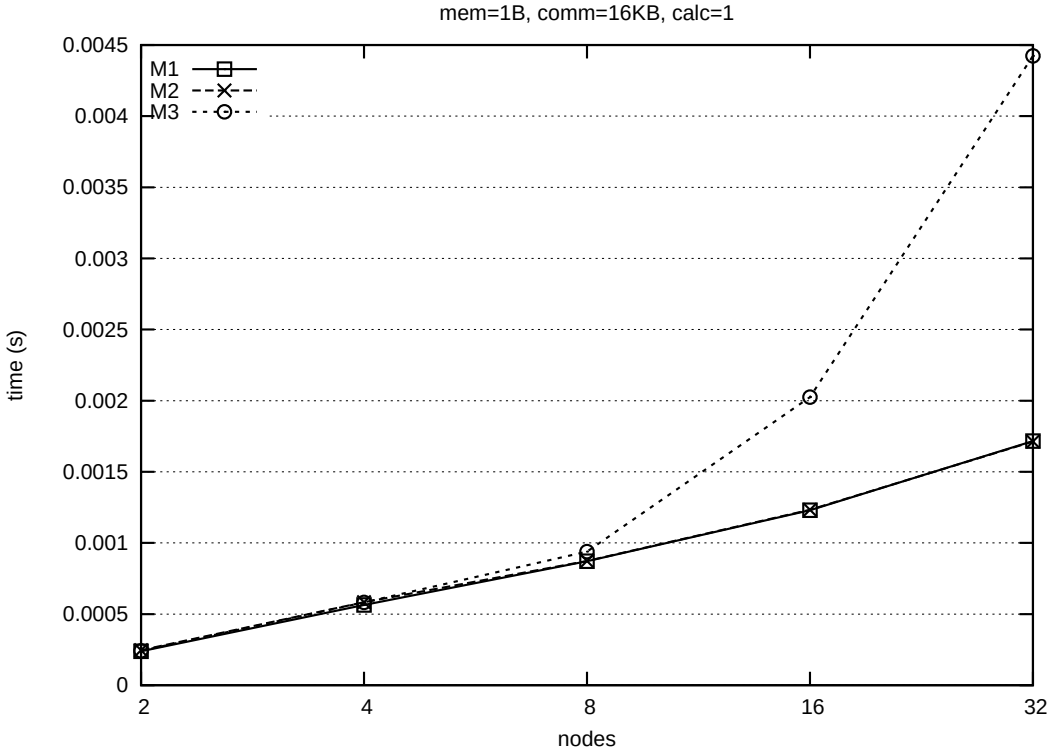

Figure 6. Scalability with 16KB read/write and no computation.

are multiple sockets with input when *poll()* is called. This reduces the number of calls and therefore the total overhead of *poll()*.

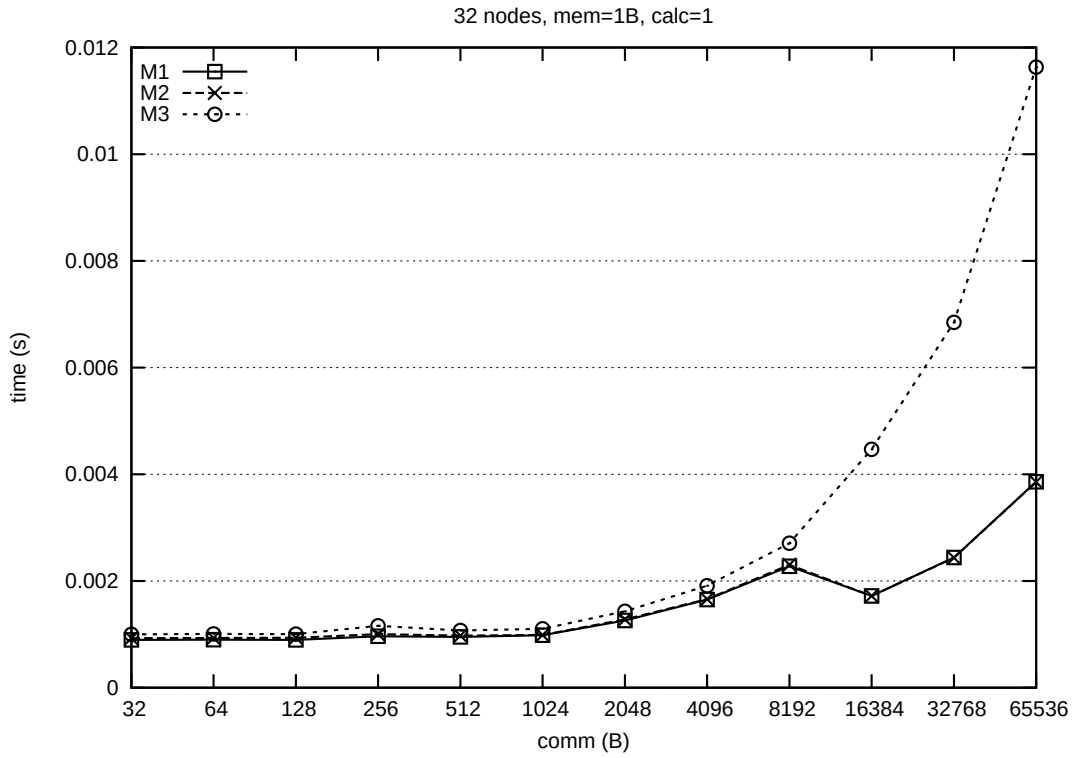

Figure 7. Different read/write sizes and no computation.

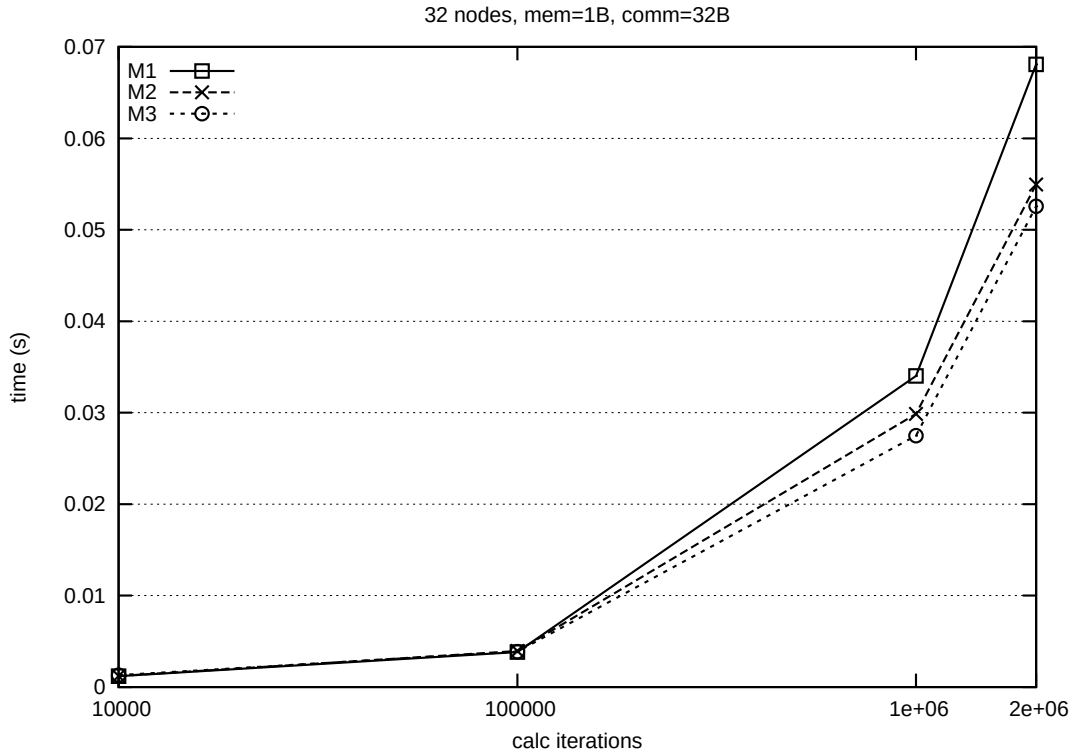

Figure 8. Different computation loads and 32B read/write.

4.3.2. CPU-bound

Figures 8 and 9 show how the models perform with different levels of computation in the clients. Remember that the actual number of iterations is random, but the displayed values are maximums. As expected M1 does not perform well when computation is increased, while M3 becomes the best model. The long computation periods spread communication events in

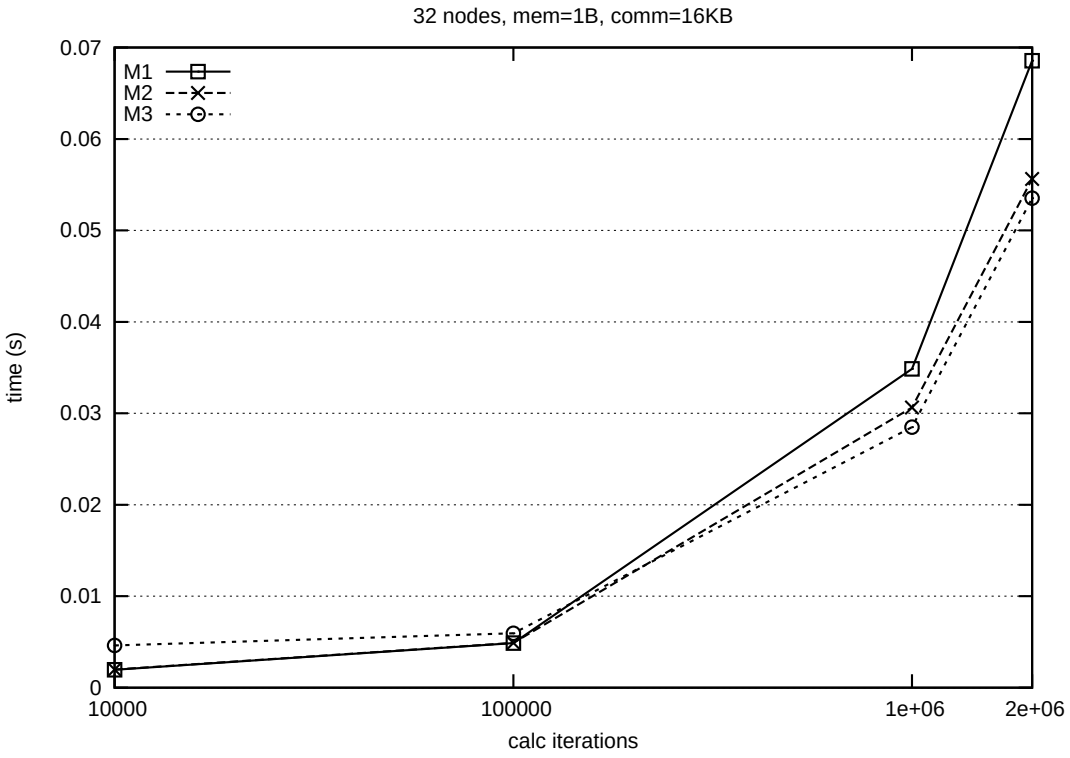

Figure 9. Different computation loads and 16KB read/write.

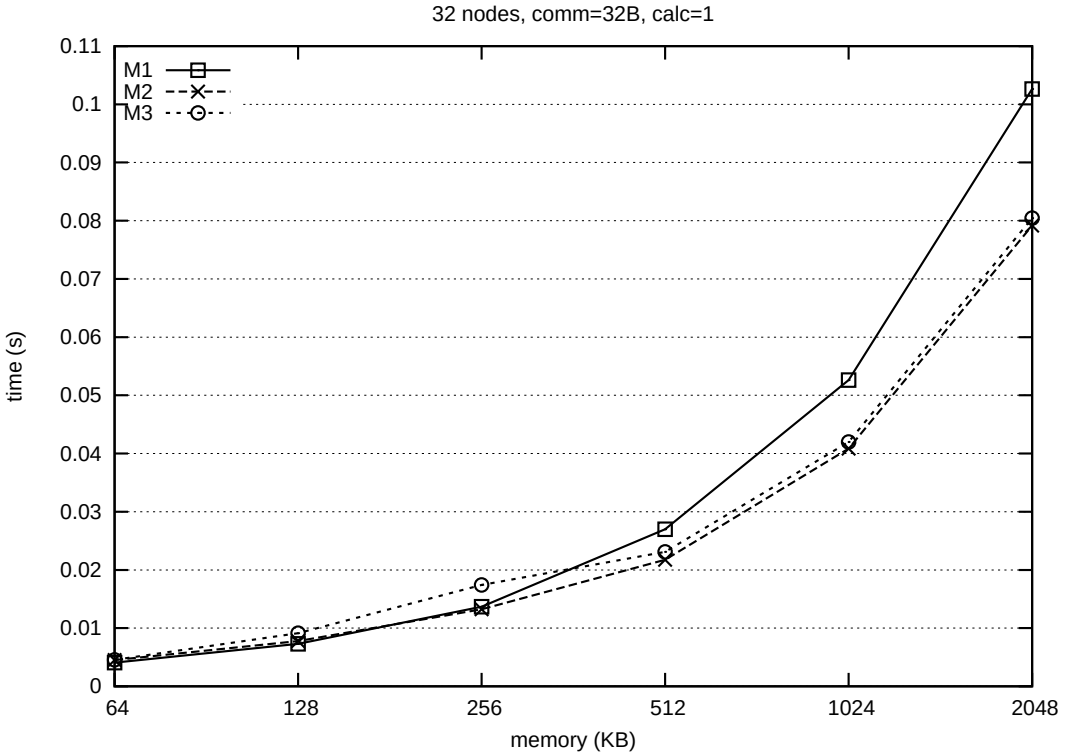

Figure 10. Client memory utilization test with 32B read/write.

time. The short wakeup path in M3 benefits from this, while the overhead of *poll()* is not amortized by handling multiple events per call. Consequently, M2 is slower than M3, but this might be resolved by using a more scalable monitoring method such as *epoll*.

When the computations touch memory M2 wins, while M3 now becomes second best (Figure 10). This is presumed to be caused by the larger working set of M3.

5. Related Work

Much work addresses TCP kernel interfaces [10,8,11,7] and revolves around monitoring multiple sockets. The general conclusion is that the performance of event-based interfaces are superior to threading. For CPU-bound workloads threading is needed, though. This is in line with the findings of this paper, because the best overall model (M2) combines event-based I/O and threads.

TCP communication latency hiding by overlapping communication with computations is explored in [12,13]. While the advantages of this overlapping are clear the evaluation is very limited. Only two nodes is used and the applications have well defined I/O and CPU-bursts.

In [14] an MPI [3] implementation that uses separate communication and computation threads is compared with a single-threaded implementation. These implementations correspond to models M2 and M1 respectively and the results are similar.

6. Conclusions

Various ways to handle TCP input in high performance distributed systems have been evaluated. This was done for a specific case where nodes act as both client and server. In this context a node is a Unix user-process that uses the kernel TCP API for communication.

Three input models with different ways of using threads were evaluated. The exact details of these models can be found in section 3, but this list gives a short summary:

M1: A single thread handling all work.
M2: A dedicated thread handling all TCP input and a client thread.
M3: A thread per TCP socket and a client thread.

The overall winner of the three models is M2. In cases where M1 or M3 are better, M2 is 10% slower at most. M1 wins in I/O-bound tests, because the single thread in this case only has to handle input events. On the other hand, it is the worst model in CPU-bound tests. M3 only wins in CPU-bound tests with low memory utilization. The short input wakeup path and large memory working set (thread state) of M3, is believed to be the reason for its effectiveness in this special case.

M1 and M2 were implemented using the *poll()* system call for socket event monitoring. More scalable methods such as *epoll* were not available on the test platform. Using such methods should shorten the wakeup path in these models.

A hybrid between M1 and M2 would be an interesting subject of further research. When the client is waiting for input it might as well handle input events. At the time the input it is waiting for becomes available, it can start using it immediately without doing a context switch.

References

[1] J. Yoakum L. Ong. *RFC 3286: An Introduction to the Stream Control Transmission Protocol (SCTP)*, 2002.
[2] S. Floyd E. Kohler, M. Handley. *RFC 4340: Datagram Congestion Control Protocol (DCCP)*, 2006.
[3] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.
[4] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.
[5] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.
[6] Brian Vinter. *PastSet: A Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, Faculty of Science, University of Troms, Norway, 1999.

[7] Dan Kegel. *The C10K problem*, 2004. http://www.kegel.com/c10k.html.
[8] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. *Proceedings of 6th Annual Linux Symposium*, 2004.
[9] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux. *White Paper, Red Hat, Fevereiro de*, 2003.
[10] J. Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference, January*, 1996.
[11] J. Lemon. Kqueue: A generic and scalable event notification facility. *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.
[12] Volker Strumpen and Thomas L. Casavant. Implementing communication latency hiding in high-latency computer networks. In *HPCN Europe*, pages 86–93, 1995.
[13] Volker Strumpen. Software-based communication latency hiding for commodity workstation networks. In *ICPP, Vol. 1*, pages 146–153, 1996.
[14] S. Majumder, S. Rixner, and V.S. Pai. An Event-driven Architecture for MPI Libraries. *Proceedings of the Los ALamos Computer Science Institute Symposium (LACSI'04),October*, 2004.