Combining Multi-Threading with Asynchronous Communication

– A Case Study with DSM-Threads using Myrinet via BIP and Madeleine –

Thomas Röblitz and Frank Mueller
Humboldt University Berlin, Institut f. Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de   phone: (+49) (30) 2093-3011   fax:-3010

Abstract

This paper contributes a study on integrating communication middleware into a multi-threading environment. It addresses problems rooted in blocking and provides solutions to build asynchronous communicating mechanisms on top of synchronous ones. The paper details different strategies for handling multi-threaded concurrency in both an event-signalling framework and a polling fashion, depending on the underlying message layer. It motivates the advantages of a single reception point for prioritized multi-threading and discusses potential for zero-copy overhead on message reception. Implementation details are given for a case study with DSM-Threads, a distributed execution environment with per node multi-threading, adapted to use Madeleine as an abstraction layer for communication and BIP on the lower level to utilize Myrinet. The study underlines strengths and weaknesses of the layered components and gives general guidelines for approaching similar efforts of communication abstraction.

1 Introduction

In recent years, multi-threading has received increased attention as a means to hide latencies and exploit shared-memory multi-processors (SMPs), which lead to standardizations such as POSIX Threads (Pthreads) [10] and incorporation of threading into main-stream languages such as Java. At the same time, network communication has seen remarkable improvements resulting in higher bandwidth and lower latency connections, *e.g.*, via Myrinet, SCI and Gigabit Ethernet. These trends have changed the field of supercomputing. Applications formerly dedicated to supercomputing have been restructured to execute in unison within a cluster of dedicated workstations. The advent of cluster computing emphasizes the need to combine the trends of multi-threading and high-efficiency communication. In particular, processor speeds are still advancing faster than improvements in the networking area. Hence, communication is likely to remain a bottleneck in distributed computing. Multi-threading may hide latencies imposed by communication. In addition, asynchronous communication allows computation to progress without waiting for message reception.

This paper addresses these trends by describing different approaches to combine multi-threading, distributed execution and message-passing communication within existing frameworks. A case study details the efforts of adapting a distributed shared memory framework, DSM-Threads [7], to adhere to communication abstraction provided by Madeleine [1] and BIP [9] for Myrinet. Several issues are raised, ranging from conflicts in control for distributed execution over different message reception paradigms to methods of integrating thread-safe message passing for asynchronous communication. Calls are considered thread-safe (or MT-safe: they block the calling thread, not the process).

DSM-Threads is a runtime system to support distributed threads with a distributed shared virtual memory. Applications that adhere to the Pthreads standard rely on shared memory and thus experience their best performance on SMPs. The programming model of these applications, however, contains inherent parallelism that is not only limited to SMPs but can readily be exploited on a distributed system with shared virtual memory, such as DSM-Threads. Hence, DSM-Threads supports an API which strongly resembles that of POSIX Threads. It supports scalability through a variety of synchronization and memory coherence protocols that strictly follow a decentralized approach, as opposed to a client-server paradigm where a server may present a bottleneck. Communication relies on message passing with point-to-point connections. It tolerates asynchronous communication on sends and allows out-of-order message passing. In addition, nodes in the distributed environment may itself be multi-threaded.

Madeleine is a framework for message passing that supports TCP/IP, BIP, VIA and SCI. Besides its task to abstract from actual network interfaces, it also provides distributed execution and supports multi-threading over a non-standard thread package, Marcel.

BIP is a fast interface to the Myrinet network architecture that supports various methods for communication to support de-facto standards as MPI or PVM.

2 Overview

This section gives an overview of the different software components, *i.e.*, starting with drivers for networking on the lowest level through various levels of middleware for communication abstraction and concurrent execution up to a distributed runtime system. Figure 1 depicts the components for our sample study but the approach can be generalized to arbitrary systems with the same design goals where different middleware components may be used. On the top level, DSM-

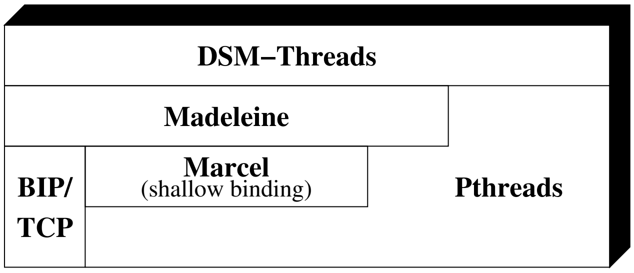

Figure 1. Layering of Software Components

Threads provides a distributed runtime system. DSM-Threads also happens to support distributed shared memory but this is not a requirement for this study, *i.e.*, we mostly abstract from the DSM features in the following. The distributed runtime requires an underlying environment for per-node multi-threading (Pthreads) and a message-passing framework. Since message passing has to be mapped onto a variety of network architectures, which are subject to constant changes and improvement in bandwidth and latency, it is imperative to choose a portable and extensible approach for supporting these network architectures. In this study, Madeleine was chosen as an intermediate layer since it supports a variety of network architectures and standards, such as Ethernet via TCP, Myrinet via BIP, SCI and VIA. Our study mostly emphasizes the usage of BIP and TCP in this context. Madeleine also provides a common interface to the upper layers, which enhances their portability.

Modifications within DSM-Threads included the handling of communication, which was mapped onto the Madeleine API, both on the sending and receiving side. Table 1 depicts the mapping from a TCP-oriented interface to the corresponding Madeleine routines. Madeleine also supports incremental sends and receives of messages, which can be used for zero-copy overhead by directly specifying the origin and destination within memory for message sends and receptions, respectively. This aspect will be detailed later on.

Madeleine is thread safe in the sense that access

TCP/IP-based	MADELEINE
`accept`	`mad_receive`
`read`	`mad_unpack_byte`
`close` (read)	`mad_recvbuf_receive`
`open`	`mad_sendbuf_init`
`write`	`mad_pack_byte`
`close` (write)	`mad_sendbuf_send`

Table 1. Mapping TCP/IP onto Madeleine

from multiple threads within one node to the network adapter is arbitrated. For this purpose, a non-standard thread package, Marcel, is utilized. Our work builds on thread packages adhering to the Pthreads standard. Hence, we decided to replace Marcel by a shallow binding package that maps a subset of the Marcel API onto Pthreads, thereby allowing Madeleine to remain unchanged with regard to its reliance on Marcel.

3 Distributed Execution

The upper-most component of the layers (DSM-Threads) provides a framework for concurrent and distributed execution in addition to enhanced runtime support, *e.g.*, to guarantee memory consistency. In this model, per-node multi-threading can be used to hide latencies and exploit SMPs within one node, which is mapped onto Pthreads. In addition, distributed execution allows dynamic creation of remote processes that also adhere to the shared-memory programming model due to the underlying runtime support for distributed shared memory. Distributed execution, as depicted in Figure 2(a), may result in one or multiple spawned processes per remote node, *i.e.*, the user can control the distribution scheme explicitly or he relies on the DSM runtime to perform this task for him. Multiple processes for one node may also help to hide latencies and exploit SMPs. This may be particularly attractive when the underlying Pthreads implementation itself only supports uni-processors (*e.g.*, FSU Pthreads [8]) but the operating system handles SMPs at the process level. Furthermore, new execution nodes can be added dynamically.

Both BIP and Madeleine actually provide two distinct concepts within one package: message-passing communication and distributed execution. These distinct concepts are tightly coupled, which makes it difficult to extract one without the other. At the same time, the upper-most layer (DSM-Threads) also provides a framework for distributed execution, as detailed above. On one hand, it would be preferable to separate distinct concepts, such as communication and execution, from one another. On the other hand, it was intended to rely on only minimal modifications of

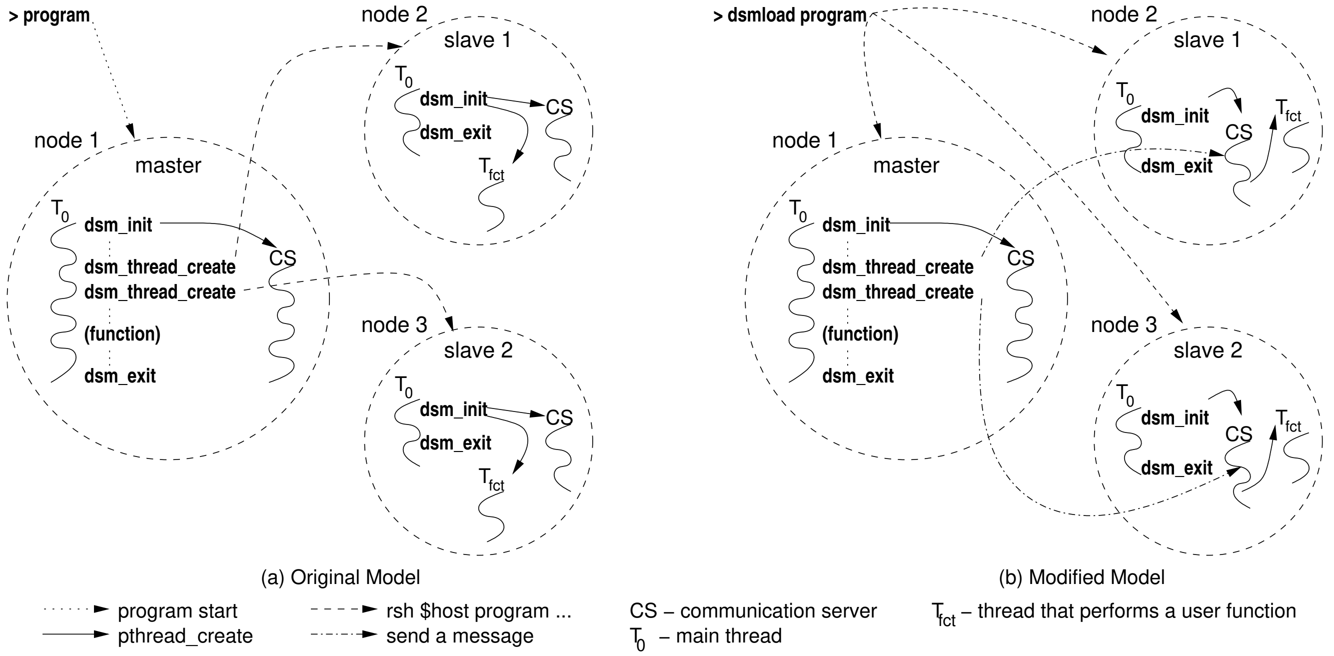

Figure 2. Distributed Execution Models for DSM-Threads

lower-level components. Hence, it was decided to adjust DSM-Threads to adhere to the distributed execution model of Madeleine and BIP. Finally, the number of nodes for distributed execution is statically regulated, *i.e.*, remote processes are created at initiation time. No additional nodes can be added later on.

Figure 2(b) depicts the modified DSM-Threads model for distributed execution. It still provides the same API to the user, *i.e.*, the user is given the option to distribute remote execution explicitly or the runtime handles this task implicitly. However, specifications of target nodes are only treated as hints. At initiation time, the DSM runtime system instructs Madeleine (and thereby BIP) to create processes on a static set of nodes. During runtime, a user request for remote execution simply results in communication with a target node to create a new thread on this node within the existing process. Hence, it is no longer possible to spawn multiple processes per node. Instead, such user requests result in clustering of a set of threads within the dedicated single process on this node. There are no impacts on the semantics of the execution model since DSM-Threads assumes a shared-memory programming paradigm. But there may be lost opportunities to exploit SMPs on the OS level when a Pthreads implementation does not already support SMPs.

4 Communication

This section contrasts different approaches to realize message-based communication with regard to several constraints. The latency and bandwidth limitations of today's systems still pose a problem for contemporary frameworks that support distributed execution. Although considerable advances both in latency and throughput have been made, the network generally remains the bottleneck in distributed environments. Hiding latencies by multi-threading may help but cannot eliminate the problem. Another option to improve the situation is to improve the responsiveness of a system, *i.e.*, to ensure that when an important message is received, it will be handled right away.

4.1 Direct and Indirect Communication

Figure 3(a) depicts a direct communication scheme. A message sent from a thread T1 on node P1 is relayed through a port or channel directly to the receiving thread T2 on P2. Each receiving thread utilizes a private port / channel for communication. This model relies on MT-safe communication calls. It can be realized for TCP/IP ports as well as BIP channels. This seemingly intuitive scheme may not always be the best choice, as detailed in the following.

Per node multi-threading and improved responsiveness on reception may sometimes be conflicting goals. In particular, multi-threading under a strict prioritized scheduling paradigm, such as given by Pthreads, may influence the communication behavior. This may result in situations where a lower priority thread is preempted while receiving a message, thereby potentially blocking the communication medium for consecutive higher priority threads for an indefinite period of time. Hence, a message sent for a higher priority thread cannot be re-

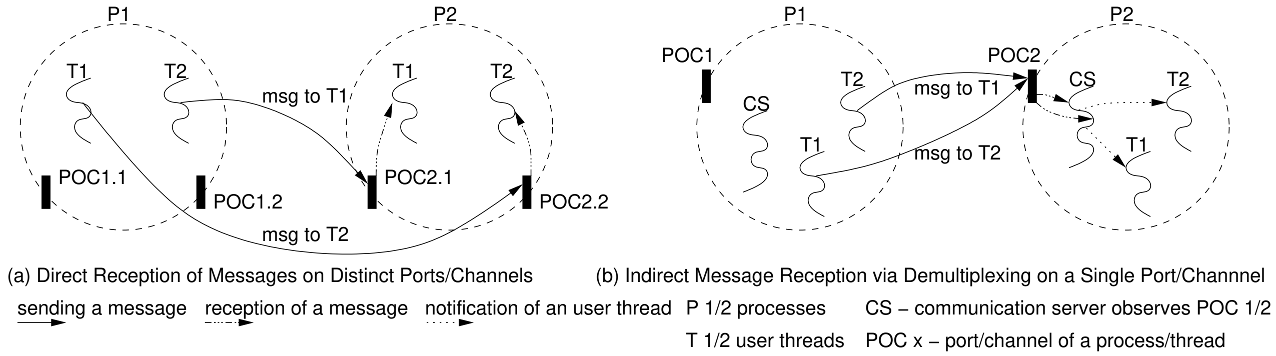

Figure 3. Connection Models for Message Passing

ceived before the lower priority thread regains control and finishes his message reception. As a result, execution on the critical path associated with a higher priority thread may be stalled, which inflicts worse overall utilization and causes longer user response times.

One solution to avoid indefinite blocking of communication devices would be to use priority inheritance [10] to boost the priority of a blocked thread when contention for communication by higher priority threads exists. In general, not all problems will be solved by priority inheritance since messages may sometimes be sent via a number of hops through various nodes. For multi-hop messages, the priority of the receiver is not known by intermediate nodes. Hence, priority inheritance may not solve the problem.

DSM-Threads uses message forwarding to support various protocols for synchronization and memory consistency, which results in multi-hop messages. A different scheme was adopted to avoid indefinite blocking on communication devices, as depicted in Figure 3(b). A dedicated thread, the communication server, handles all incoming messages on a single port or channel. Messages are demultiplexed within the node triggering synchronization to wake up the true recipient. The communication server executes at a higher priority than user threads, which ensures the responsiveness of the system. The communication medium will never be blocked indefinitely as incoming messages are received immediately after the previous messages was handed off to the proper thread by the communication server. This scheme was adopted by DSM-Threads for TCP/IP ports as well as BIP channels, both supported via the common Madeleine abstraction layer.

4.2 MT-Safe Message Passing

Operations for communication, namely sending and receiving, may block execution. Such synchronous communication is required by the receive operation while the send directive may operate synchronously or even asynchronously (without blocking). It is required that synchronous communication be MT-safe (only the calling thread is blocked, not the process).

Standard communication interfaces provided by an operating system typically provide MT-safe communication interfaces and even user-level Pthreads implementations often support this feature. This allows communication through TCP/IP to be directly used. Internally, notification upon message reception is implemented via event (signal) notification, which results in resumption of thread execution. Figure 4 depicts this situation. A signal handler is registered at program start. Subsequent receive calls block the current thread. When a message arrives, the handler supplies the message to the thread and reactivates its execution. The thread then resumes with the processing of the message. The kernel (or the Pthreads implementation) hides these details of signal delivery and execution handling from the user.

Non-standard communication interfaces, *e.g.*, add-on drivers such as BIP for Myrinet, often do not support asynchronous notification of message arrivals via signals. Interfaces for message reception either block the process or probe for arrived messages. A multithreaded environment has to ensure that only the current thread is blocked upon message reception. Hence, probing for message arrival can be used in a polling scheme to emulate event notification by signals. In Figure 5, a thread receives a message by first supplying a buffer and receives a handle on some condition variable. While the test indicates that the probe for new messages was negative, the thread voluntarily suspends for a short period of time before the test is repeated. Once the message arrives, the next probe causes the thread to exit the loop and then process the message.

Polling has the advantage that it can be easily implemented over most communication libraries. Sleep-based polling does not avoid the overhead of signalling since a timer event is generated upon expiration of the sleep duration. Since polling may result in repeated suspension and resumption of thread execution, its overhead due to context switches and signals may

```
registerHandler( ReceiveMessage )
      :
receive( buffer )                      suspends execution of the thread
   ┌─ Handler ──────────────────
   │ ReceiveMessage                    msg available on the network
   │   process message basically       copy msg to buffer, etc.
   │   return                          resume waiting thread
   └────────────────────────────

process message
```

Figure 4. Message Reception after Event Notification

```
handle := init( buffer )
while(!test( handle ))             test != 0 ──► msg available in buffer
  sleep( x ms )                    wait x milliseconds; blocks thread;
                                   other threads may become active

process message
```

Figure 5. Message Reception via Polling

exceed that of event signalling upon message reception.

The polling scheme was adopted for the communication server thread in DSM-Threads. This server thread already executes at a higher priority than user threads, which guarantees that messages will be consumed at the communication device as soon as they arrive, given that the prior message has already been consumed in the same manner. This ensures responsiveness of the multi-threaded node at the price of the polling overhead for computationally intensive applications. Recall that we assume the communication medium to represent the bottleneck in distributed execution. Hence, trading the polling overhead for increased responsiveness seems promising. We made a similar experience when porting the communication interface of DSM-Threads to SCI. Here, we used the Common Message Layer (CML) [3, 2]. Again, only message probing but not signal notification was supported to check for receptions. We also used the polling approach in this case to integrate CML with a multi-threading environment.

4.3 Partial Message Processing

The communication server of DSM-Threads currently receives an entire message. The server thread then decides if a simple or a more complex operation is required to handle the message. Simple actions, such as message forwarding to another node, are performed by the server thread itself. More complex messages will be handed off to a set of worker threads operating at lower priority. Handing a message to a worker generally implies that a copy of the message data be generated, as depicted in Figure 6(a).

An alternate approach would be to only receive the message header within the communication server but leave the remaining message to a worker thread, as seen in Figure 6(b). This partial (incremental) reception of a message also provides opportunities for zero-copy overhead when large data is transfered between nodes. Page based DSM systems, for example, can benefit by placing entire pages in the correct memory location as part of their consistency protocols. BIP actually provides asynchronous communication, which is supported by partial message processing primitives within Madeleine as thread-blocking operations. Due to the underlying scheduling of Marcel, other threads continue to execute and may potentially utilize the CPU for some time before the blocked thread is reactivated and gets a chance to receive the message. Hence, Madeleine over Marcel may inflict longer response times than our solution of DSM-Threads over prioritized Pthreads. In Madeleine/Marcel, reception may be subject to voluntary suspension of other threads while DSM-Threads/Madeleine/Pthreads ensures responsiveness through the polling interval.

5 Modifications to the Communication

Several modifications had to be incorporated into the different software layers to adapt to the new communication schemes. Changes were kept to an absolute minimum within a localized portion of the software components. Most changes were incorporated within selected places in Madeleine or as separate modules to map equivalent functionality.

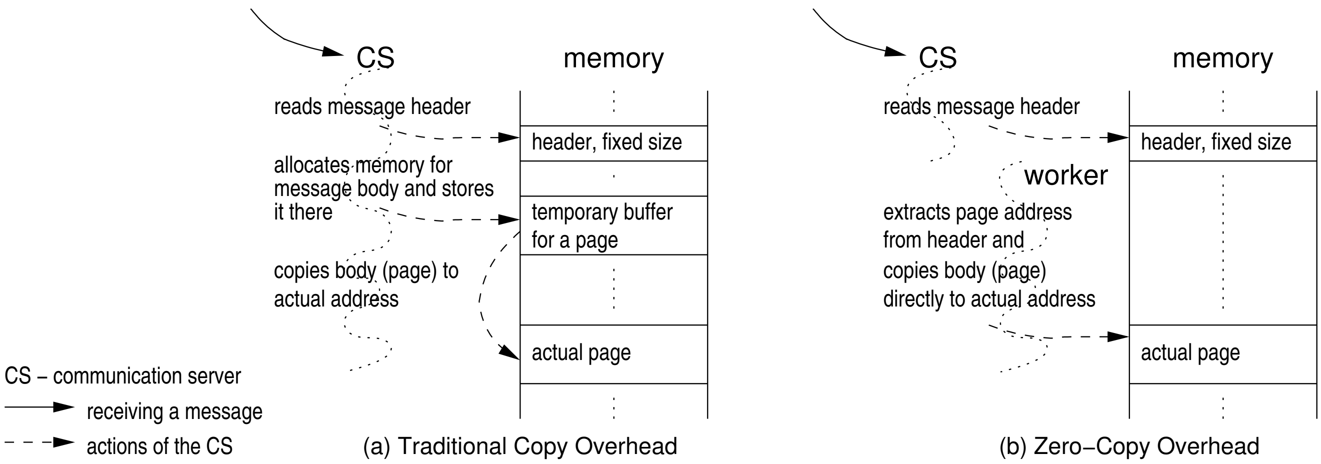

Figure 6. Message Unpacking Models

5.1 Replacing Marcel

Madeleine's usage of the Marcel thread package was simply replaced by a shallow binding to the equivalent Pthreads functionality provided as a separate module (see Table 2 for the respective interfaces and types).

Marcel	Pthreads
Routines	
lock_task	pthread_mutex_lock
unlock_task	pthread_mutex_unlock
marcel_key_create	pthread_key_create
marcel_setspecific	pthread_setspecific
marcel_getspecific	pthread_getspecific
marcel_mutex_init	pthread_mutex_init
marcel_mutex_lock	pthread_mutex_lock
marcel_mutex_unlock	pthread_mutex_unlock
marcel_givehandback	sched_yield
tmalloc	adopted from MARCEL.
tfree	adopted from MARCEL.
marcel_select	select
marcel_sem_init	sem_init (uses lock/cond)
marcel_sem_P	sem_p (uses lock/cond)
marcel_sem_V	sem_v (uses lock/cond)
Data Types	
marcel_mutex_t	pthread_mutex_t
marcel_mutexattr_t	pthread_mutexattr_t
marcel_key_t	pthread_key_t

Table 2. Shallow Binding: Marcel to Pthreads

Lock and unlock utilize a global lock in conjunction with the corresponding Pthreads primitives. Madeleine uses a global semaphore to arbitrate concurrent communication requests. The corresponding semaphore operations `lock_task` and `unlock_task` could be replaced by a global lock for mutual exclusion. Other semaphore operations were resembled by a combination of Pthreads locks and condition variables.

5.2 Adapting Madeleine's BIP-Interface

Consider Table 2 again. The Marcel interfaced `marcel_givehandback` provides similar functionality as the POSIX `sched_yield` interface. However, in the presence of strict priority scheduling, suspension only affects threads at the same priority level. If the communication server tried to yield to another thread, it would not find another thread at the same priority level and simply continue to execute. This problem was addressed by implementing polling in the previously described fashion. Suspension for a limited duration allows lower priority threads to gain control to progress with their execution.

Further extensions comprise mainly two areas, sending large messages and the hand-shaking mechanism of credits for BIP buffers. Madeleine uses a multi-stage model to send long messages by partitioning them into three parts depicted in Figure 7(a).

1. The sender communicates a request to the recipient as part of a short message.

2. The recipient returns an acknowledgement to signal that he is ready to receive a long message.

3. The sender then progresses by transmitting the remaining part of the actual message.

Prioritized multi-threading environments, such as DSM-Threads, rely on a single receiving thread, in our case the communication server, as detailed earlier. The multi-stage send protocol for large messages was extended to the effect that the communication server receives the acknowledgement message and reactivates the user thread through conditional synchronization, *i.e.*, a conditional signal (see Figure 7(b)). If the communication server himself sends a large message, the wait and signal operations can simply be omitted

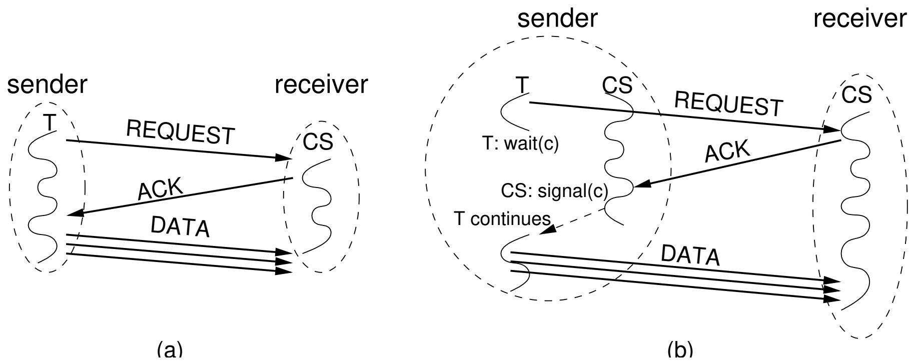

Figure 7. Models for Sending Large Messages under BIP

but the server has to remember pending partial operations. Demultiplexing of messages to the correct thread is realized through separate condition variables for each thread. Hence, signaling a specific condition only awakes one selected thread.

The second extension addresses the hand-shaking mechanism for credit handling to ensure that the limited number of buffers within BIP does not overflow. In BIP, each receiver posses a limited number of buffers per sender for storing short messages. In the beginning, a sender posses all credits for a receiver, as depicted Figure 8(a) with B_{snd} and B_{rcv} credits for sends to B and receives from B, respectively. The latter denotes the number of already used credits. When node A sends a message to B, then credits on A for B will be decremented, as seen in Figure 8(b-d). A send without proper credits simply results in blocking since it must be assumed that the receiver does not have an empty slot for short messages. In addition, used credits will be returned as piggybacks on every message, as seen in Figure 8(c). This informs the other node that a slot has become available for another message.

Sending messages in DSM-Threads was modified to the extend that (in the absence of credits) messages are stored in a message queue. This effectively results in asynchronous sends for threads on the DSM level. The message queue is later handled by workers when new credits are received. The worker is informed by the communication server about the arrival of new credits. Depending on the number of received credits, a subset of the queued messages may then be sent. Notice that DSM-Threads uses asynchronous communication protocols in general for synchronization and consistency handling on the DSM level, which facilitates credit hand-shaking in the described manner.

6 Related Work

Langendoen *et al.* address issues of multi-threading in the presence of interrupts and polling for messages [5]. They investigate performance issues and mechanisms for matching polling rates and message arrival rates. They conclude that signals are more appropriate for coarse parallelism while polling excels under finer granular parallelism. They also suggest to combine both approaches depending on the current state of the scheduler, *i.e.*, they suggest to use polling when no threads are ready. Our work differs in that we aim to minimize modifications to existing components. Neither the thread scheduling (Pthreads), which may even be inside the operating system, nor the communication layer (BIP) was modified in our work.

Maquelin *et al.* also suggest to combine polling and interrupt handling for communication but on a hardware level [6]. Interrupts are only generated if the network interface realizes that polling may not result in effective responsiveness. Results show that this approach outperforms traditional polling in terms of responsiveness while achieving comparable performance overhead. At the same time, this overhead is lower than traditional interrupt handling. In contrast, our work was restricted to software approaches of dealing with communication issues.

Itzkovitz *et al.* discuss the merits of interrupts and polling on Windows NT for Millipede [4]. They suggest that the combination of polling, multi-threading and asynchronous communication may neither achieve the best responsiveness, nor utilization, nor the lowest number of context switches. They enhanced Fast Messages to implement signal notification. Our work focuses on portability and extensibility with regard to existing communication components. Hence, we did not enhance BIP by signal notification. Future work may include such an approach to assess if the results from NT can be generalized to other environments.

7 Conclusion

We presented our experience on integrating communication middleware into a multi-threading envi-

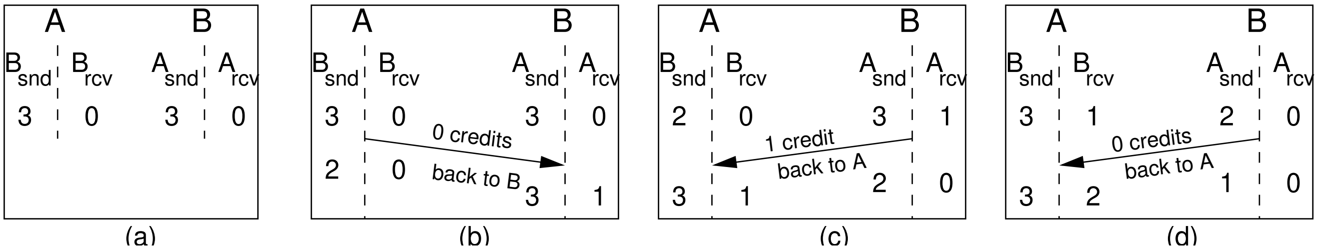

Figure 8. Credit Mechanism

ronment. Several problems rooted in blocking were addressed and a number of solutions were suggested to build asynchronous communicating mechanisms on top of synchronous ones. The paper detailed different strategies for handling multi-threaded concurrency in both an event-signalling framework and a polling fashion, depending on the underlying message layer. It motivated the advantages of a single reception point for prioritized multi-threading and discussed potential for zero-copy overhead on message reception. Implementation details were given for a case study with DSM-Threads, a distributed execution environment with per node multi-threading, which was adapted to use Madeleine as an abstraction layer for communication and BIP on the lower level to utilize Myrinet. In summary, a clear separation of communication and execution paradigms would have facilitated the port but may have compromised the flexibility of BIP and Madeleine. In addition, event notification through signals upon message receptions would have facilitated the port. It would have also have provided a more responsive scheme of message handling with comparable overhead in the best case and less overhead in the worst case. The study underlined strengths as well as weaknesses of the involved layering components and gave general guidelines for approaching similar efforts of communication abstraction.

References

[1] Luc Bougé, Jean-Fran cois Méhaut, and Raymond Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT '98)*, pages 240–247, ENST, Paris, France, October 1998. IFIP WG 10.3 and IEEE.

[2] M. Eberl, W. Karl, M. Leberecht, and M. Schulz. Eine Software-Infrastruktur für Nachrichtenaustausch und gemeinsamen Speicher auf SCI-basierten PC-Clustern. In *Cluster Computing Workshop*, 1999.

[3] B. Herland and M. Eberl. A common design of PVM and MPI using the SCI interconnect. final report for task a 2.11 of ESPRIT project 23174, University of Bergen and Technical University of Munich, October 1997.

[4] Ayal Itzkovitz, Assaf Schuster, and Yoram Talmor. Harnessing the power of fast, low latency, networks for software dsms. In *ACM Workshop on Software Distributed Shared Memory*, Rhodes, Greece, June 1999.

[5] K.G. Langendoen, J. Romein, R.A.F. Bhoedjang, , and H.E. Bal. Integrating polling, interrupts, and thread management. In *Symposium on the Frontiers of Massively Parallel Computation*, pages 13–22, Annapolis, Maryland, October 1996.

[6] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, , and Xin-Min Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. In *International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 1996.

[7] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[8] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[9] Loïc Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485. Springer-Verlag, April 1998.

[10] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].