Designing a Common Communication Subsystem*

Darius Buntinas and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
{buntinas, gropp}@mcs.anl.gov

Abstract. Communication subsystems are used in high-performance parallel computing systems to abstract the lower network layer. By using a communication subsystem, an upper middleware library or runtime system can be more easily ported to different interconnects. By abstracting the network layer, however, the designer typically makes the communication subsystem more specialized for that particular middleware library, making it ineffective for supporting middleware for other programming models. In previous work we analyzed the requirements of various programming-model middleware and the communication subsystems that support such requirements. We found that although there are no mutually exclusive requirements, none of the existing communication subsystems can efficiently support the programming model middleware we considered. In this paper, we describe our design of a common communication subsystem, called CCS, that can efficiently support various programming model middleware.

1 Introduction

Communication subsystems are used in high-performance parallel computing systems to abstract the lower network layer. By using a communication subsystem, an upper middleware library or runtime system can be ported more easily to different interconnects. By abstracting the network layer, however, the designer typically makes the communication subsystem less general and more specialized for that particular middleware library. For example, a communication subsystem for a message-passing middleware might have been optimized for transferring data located anywhere in a process's address space, whereas a communication subsystem for a global address space (GAS) language might have been better optimized for transferring small data objects located in a specially allocated region of memory. Thus, the communication subsystem designed for a GAS language cannot efficiently support the message-passing middleware because, for example, it cannot efficiently transfer data that is located on the stack or in dynamically allocated memory.

Despite their differences, communication subsystems have many common features, such as bootstrapping and remote memory access (RMA) operations. In

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

[1] we analyzed the requirements of various programming model middleware and the communication subsystems that support them. We found that although there are no mutually exclusive requirements, none of the existing communication subsystems can efficiently support the programming-model middleware we considered. In this paper, we describe our design of a common communication subsystem, called CCS, that can efficiently support various programming-model middleware. We specifically targeted CCS to efficiently support the requirements of MPICH2 [2,3], the Global Arrays (GA) toolkit [4,5], and the Berkeley UPC runtime [6,7]; however, we believe that CCS is general enough to efficiently support any message-passing, global address space, or remote-memory middleware.

The rest of this paper is organized as follows. In Section 2 we briefly describe the critical design issues necessary to support the various programming models. In Section 3 we present our design for a common communication subsystem. In Section 4 we show performance results from our preliminary implementation of CCS. In Section 5 we conclude and present future work.

2 Design Issues for Communication Subsystems

In this section, we briefly describe the important issues for designing a common communication subsystem. These design issues are covered in more detail in [1]. We divide the design issues into required features and desired features. A required feature is a feature that, if lacking, would prevent the communication subsystem from effectively supporting a particular programming model. Desired features are features that, when implementing a programming model on top of the communication subsystem, make the implementation simpler or more efficient.

2.1 Required Features

Remote Memory Access Operations. RMA operations allow a process to transfer data between its local memory and the local memory of remote process without active participation of the remote process. RMA operations are important for global address space and remote-memory programming models, as well as for message-passing applications that have irregular communication patterns.

In order to allow better overlap of communication and computation, non-blocking RMA operations should be provided. A mechanism is then needed to check whether the operation has completed.

MPI-2 RMA support. In order to support MPI-2 [8] active-mode RMA operations, the communication subsystem must be able to perform RMA operations between any memory location in the process's address space. In order to support passive-mode RMA operations, the communication subsystem need only be able to perform RMA operations on memory that has been dynamically allocated using a special allocation function.

GAS language and remote-memory model support. GAS language and remote-memory model runtime systems need to be able to perform concurrent conflicting RMA operations to the same memory region. Similarly, they require

the ability to perform local load/store operations concurrently with RMA operations, possibly to the same memory location. While the result of such conflicting operations may be undefined, the communication subsystem must not consider it an error to perform them. RMA operations also must be very lightweight, since typical RMA operations in these programming models are single-word operations.

Efficient Transfer of Large MPI Two-Sided Messages. MPI and other message-passing interfaces provide two-sided message passing, where the sending process specifies the source buffer, and the receiving process specifies the destination buffer. Typically, in message-passing middleware, large data is transferred by using a rendezvous protocol, where one process sends the address of its buffer to the other process, so that one process has the location of both the source and destination buffers. Once one process has the location of both buffers, it can use RMA operations to transfer the data. In MPI, the source and destination buffers can be located anywhere in the process's address space. In order to support transferring large two-sided messages in this way, the communication subsystem must be able to perform RMA operations on any memory location in the process's address space.

2.2 Desired Features

Active Messages. Active messages [9] allow the sender to specify a handler function that is executed at the receiver when the message is received. This function can be used, for example, to match an incoming message with a preposted receive in MPI or to perform an accumulate operation in Global Arrays.

In order to support multiple middleware libraries at the same time, active messages from one middleware library must not interfere with those of another middleware library. One solution is to ensure that each library uniquely specifies its own handlers.

In-Order Message Delivery. In-order message delivery is a requirement for many message-passing programming models. If the communication subsystem provides this feature, the middleware doesn't have to deal with reordering messages. However, in other programming models such as GAS languages, message ordering is not required, and in some cases performance can be improved by reordering or coalescing messages. A common communication subsystem should be able to provide FIFO ordering when it is required, and allow messages to be reordered otherwise.

Noncontiguous Data. Programming model instances such as MPI and Global Arrays have operations for specifying the transfer of noncontiguous data. Furthermore, modern interconnects such as InfiniBand (IBA) [10], support noncontiguous data transfer. Hence, a common communication subsystem needs to support the transfer of noncontiguous data in order to take advantage of such functionality.

Table 1. Feature summary of the communication subsystems.

	RMA operations	MPI-2 active-mode RMA	MPI-2 passive-mode RMA	GAS language support	Transfer of large MPI messages	Active messages	In-order message delivery	Noncontiguous data*	Portability
ARMCI	•			•				V, S	•
GASNet	•		•	•		•			•
LAPI	•	•	•	•	•	•		V	
Portals	•	•	•	•	•		•		•
MPI-2	•	•	•		•		•	V, S, B	•

* V = I/O vector; S = strided; B = blockindexed

2.3 Feature Support by Current Communication Subsystems

In [1] we examined several communication subsystems and evaluated how well each addresses the features described above. Table 1 summarizes the results. We evaluated ARMCI [11], GASNet [12], LAPI [13], Portals [14], and MPI-2 [8] as communication subsystems. We can see from this table that none of the communication subsystems we studied supports all of the features necessary for message-passing, remote-memory, and GAS language programming models.

3 Proposed Communication Subsystem

In this section we describe our design for a common communication subsystem, called CCS, that addresses the issues identified in the previous section. The CCS communication subsystem is based on nonblocking RMA operations, with active messages used to provide for control and remote invocation of operations. Active messages could be used to implement an operation to deliver the message in message passing middleware or to perform an accumulate operation in remote memory middleware.

3.1 Remote Memory Access Operations

CCS provides nonblocking RMA operations. It is intended that RMA operations be implemented by using the interconnect's native RMA operations in order to maximize performance. If an interconnect does not natively provide all or some of the required RMA operations, active messages can be used to implement the missing RMA operations. For example, if an interconnect has a native Put operation but not a Get operation, the Get can be implemented with an active message in which the handler performs a Put operation.

CCS uses a *callback* mechanism to indicate the completion of RMA operations. A callback function pointer is specified by the upper layer as a parameter to the RMA function. Then, when an RMA operation completes remotely, CCS

calls the callback function. This can be used to implement fence and global fence operations.

Because the user-level communication libraries of most interconnects require memory to be registered before RMA operations can be performed on that memory, CCS also requires memory registration. The upper layer is responsible for ensuring that any dynamically allocated memory is deregistered before it is freed. The current design is to limit the amount of memory that a process can register to the amount that can be registered with the interconnect. A future design is to lift this restriction. If the upper layer registers more memory with CCS than the interconnect can register, CCS would handle deregistering and reregistering memory with the interconnect as needed. A mechanism similar to the *firehose* [15] mechanism used in GASNet could be employed.

Registering and deregistering memory with a network library usually involve a system call, which makes them costly operations. In order to reduce the overhead of registering and deregistering memory, CCS implements a registration cache and uses lazy deregistration. CCS keeps track of which pages have already been registered, to avoid registering pages twice. CCS also does not immediately deregister memory when the upper layer calls the CCS deregistration function. Instead, CCS simply decrements the usage count and deregisters pages once the number of unused pages reaches a certain threshold. This scheme reduces the number of network library registration and deregistration calls.

CCS RMA operations can access all of the process's memory and have no restrictions on concurrent access to memory. While this feature simplifies implementing upper layers on CCS, it can impact performance on machines that are not cache coherent and on interconnects that do not have byte granularity for their RMA operations. In these cases, CCS will have to handle the RMA operations in software taking care of cache coherence and data transfer.

3.2 Efficient Transfer of Large MPI Two-Sided Messages

CCS nonblocking RMA operations are to be used for transferring large messages. CCS RMA operations are intended to be implemented by using native interconnect RMA operations to maximize throughput. Because the operations are nonblocking, the communication can be overlapped with computation.

As described in the previous section, large MPI messages are typically transferred by using a rendezvous operation. In CCS, the rendezvous operation can be performed with active messages. Once the exchange of buffer locations has been done, the data can be transferred with RMA operations. When the RMA operations have completed, another active message would be sent to notify the other side of completion.

A future design is to implement a *large data active message* operation, which would function similar to the LAPI active messages using the header handler. The large data active message would be nonblocking. The sender would specify an active message handler and a local completion handler. The active message handler would be executed at the receiver before any data has been transferred. The handler would specify the receive buffer and its local completion handler. Once the data had been transferred, the completion handlers on the sender and receiver would be called. A mechanism would be needed for the receiver to abort or delay the operation in the active message handler if it was not ready to receive the data yet.

3.3 Active Messages

We are including active messages in CCS because of the flexibility they provide to upper-layer developers. In our design, active messages are intended to be used for small message sizes, so the implementation should be optimized for low latency.

When an active message is received and the handler is executed, the handler gets a pointer to a temporary buffer where the received data resides. The handlers are responsible for copying the data out of the buffer. Noncontiguous source data will be packed contiguously into the temporary buffer. If the final data layout is to be noncontiguous, the message handler will have to unpack the data.

Depending on the implementation, the active message handlers will be called either asynchronously or from within another CCS function. CCS provides locks that are appropriate to be called from within the handler and includes a mechanism to prevent a handler from interrupting a thread.

To allow multiple upper layer libraries to use CCS at the same time, we introduce the notion of a *context*. Each separate upper layer library, or module, allocates its own context. Active message handlers are registered with a particular context. When an active message handler is registered, the upper layer provides the handler function pointer along with an ID number and the context. The ID number must be unique within that context. When an active message is sent, the context is specified along with the handler ID to uniquely identify the active message handler at the remote side.

3.4 In-Order Message Delivery

In order to support the message-passing programming model, CCS guarantees in-order delivery of active messages. However, RMA operations are not guaranteed to be completed in order. This approach allows CCS, or the underlying interconnect, to reorder messages in order to improve performance.

3.5 Noncontiguous Data

CCS supports noncontiguous data in active messages and RMA operations. CCS uses *datadescs* to describe the layout of noncontiguous data. Datadescs are similar to MPI datatypes and are, in fact, implemented by using the same mechanism that MPICH2 uses for datatypes [16]. Datadescs are defined recursively like MPI datatypes; however, datadescs do not currently store information about the native datatype (e.g., `double` or `int`) of the data. Because datadescs do not keep track of native datatypes, datadescs CCS cannot be used on heterogeneous systems, where byte-order translation would need to be done. We will address this situation in future work.

While datadescs are defined recursively, they need not be implemented recursively. In the implementation the datadesc can be unrolled into a set of component loops, rather than use recursive procedure calls that would affect performance. These unrolled representations can be used to efficiently and concisely describe common data layouts such as ARMCI strided layouts.

MPI datatypes can be implemented by using datadescs having the upper layer keep track of the native datatypes of the data. I/O vector and strided data layouts in LAPI and ARMCI can also be represented with datadescs. An implementation optimization would be to include specialized operations to create datadescs quickly from the commonly used I/O vector and strided representations in LAPI and ARMCI.

3.6 Summary of Proposed Communication Subsystem

Our proposed communication subsystem addresses all of the issues raised in the previous section. Active messages can be used by GAS language and remote-memory copy middleware for remote-memory allocation and locking operations and by message-passing middleware for message matching. Because CCS supports multiple contexts for active messages, it can be used for hybrid programming models, for example, where an application uses both MPI-2 and UPC.

CCS provides RMA operations that are compatible with MPI-2 RMA operations, as well as GAS language and remote memory copy RMA operations. CCS has primitives that can be used to implement fence and global fence operations. With the addition of a symmetric allocation function, GAS language and remote memory copy RMA support can be implemented very efficiently. The CCS RMA operations can also be used for transferring large messages in message-passing middleware.

CCS also provides in-order message delivery for active messages but does not force RMA operations to be in order. This feature allows active messages to be used for MPI-2 message-passing, while allowing RMA operations to be reordered for efficiency.

CCS supports transfer of noncontiguous data. The data layout is described in a recursive manner but can be internally represented compactly and efficiently. CCS's *datadescs* are compatible with MPI-2 datatypes. Strided and IOV data descriptions used in Global Arrays can also be efficiently represented with datadescs.

Our design of using RMA operations with active messages was inspired by LAPI and GASNet. But, as we showed in [1], LAPI and GASNet do not support all of the key features necessary to efficiently support all of the programming models we targeted. LAPI does not guarantee in-order message delivery, supports only I/O vector style of noncontiguous data, and is not portable. GASNet does not support MPI-2 active-mode RMA operations, the efficient transfer of large MPI messages, in-order message delivery, or noncontiguous data.

We note that the lack of some of the features we described does not necessarily mean that a middleware cannot be implemented over a particular communication subsystem. In fact, MPI has been implemented over LAPI [17], UPC has been implemented over MPI [7], and MPI-2 has been implemented over GASNet [2]. But the lack of these features makes these implementations less efficient and more difficult. By implementing all of the key features, CCS can efficiently support all of the programming-model middleware.

Figure 1 shows some sample code using CCS. The code sends an active message, using `CCS_amrequest()`, to another node with no data, but with the pointer and length to its local buffer as parameters to the message handler. The message handler on the receiving side calls `CCS_get()` to get the data stored in the buffer specified by the sender. When the Get operation completes CCS will call the callback function `get_callback()` specified in the call to `CCS_get()`.

4 Preliminary Performance Results

In this section we present performance results for our preliminary implementation of CCS over GM2 [18]. We performed latency and bandwidth tests on two dual 2 GHz Xeon nodes running Linux 2.4.18 and connected with a Myrinet2000 network [19] using Myricom M3F-PCI64C-2 NICs through a 16-port switch.

```
void get_callback (void *arg) {
    ++gets_completed;
}
#define NEW_MSG_HANDLER_ID 0
void new_msg_handler (CCS_token_t token, void *buffer, unsigned buf_len,
                      void *remote_buf, int remote_buflen) {
    int sender;
    CCS_sender_rank (token, &sender);
    CCS_get (sender, remote_buf, remote_buflen, CCS_DATA8, buf, buflen,
             CCS_DATA8, get_callback, 0 /* callback argument */);
}
int main (int argc, char **argv) {
    CCS_init();
    CCS_new_context (&context);
    CCS_register_handler (context, NEW_MSG_HANDLER_ID, new_msg_handler);
    buf = malloc (buflen);
    CCS_register_mem (buf, buflen);
    CCS_barrier();
    ...
    CCS_amrequest (context, other_node, NULL, 0, CCS_DATA_NULL,
                   NEW_MSG_HANDLER_ID, 2, buf, buflen);
    ...
    CCS_finalize();
}
```

Fig. 1. Sample CCS code

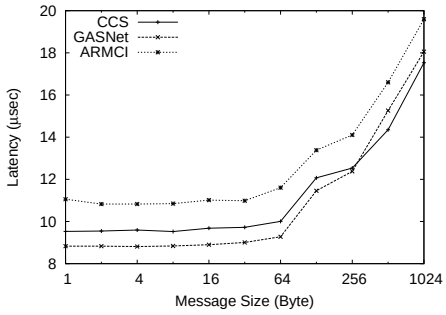

Fig. 2. Latency

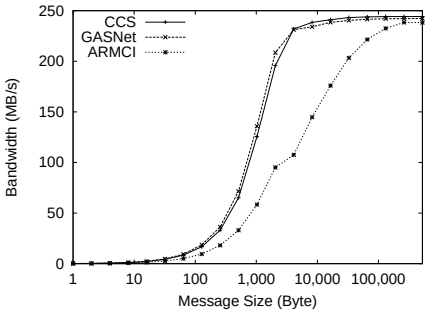

Fig. 3. Bandwidth

Figure 2 shows the latency results for CCS as well as for GASNet 1.3 and ARMCI 1.2B. For CCS and GASNet, we performed the test using active messages. Because ARMCI supports only RMA operations, we performed the test using Put. The results are averaged over 1,000 iterations. The 4-byte latency for GASNet is 8.8 μs, for CCS is 9.6 μs, and for ARMCI is 10.8 μs. We see from these numbers and Figure 2 that CCS performs better than ARMCI but not as well as GASNet.

Figure 3 shows the bandwidth results. We used nonblocking Put operations to perform the test. In this test, for each message size, we performed 10,000 Put operations, then waited for the operations to complete. We see that CCS performs better than ARMCI for all message sizes and better than GASNet for messages larger than 4 KB. For messages smaller than 4 KB, CCS performs only slightly worse than GASNet. The maximum bandwidth for CCS was 244 MBps, for GASNet was 242 MBps, and for ARMCI was 238 MBps.

The performance of CCS is comparable to the other communication subsystems. The additional functionality of CCS does not have a large impact on performance. We have not yet tuned the CCS source code for performance and expect that that with some performance tuning, we can further improve the performance of CCS.

We note that ARMCI over GM is implemented by using a server thread on each node. In ARMCI, RMA operations from remote processes are performed by the server thread. While using a server thread may affect performance compared to CCS and GASNet, it does allow RMA operation to complete asynchronously, independent of what the application thread is doing. We intend to evaluate such functionality for CCS.

5 Discussion and Future Work

In this paper we have presented our design for a common communication subsystem, CCS. CCS is designed to support the middleware libraries and runtime systems of various programming models efficiently by taking advantage of the high-performance features of modern interconnects. We evaluated a preliminary implementation of CCS and found the performance to be comparable to that of ARMCI and GASNet.

In the future, we intend to address thread safety, RMA Accumulate operations, and collective communication operations. We also intend to implement atomic remote memory operations, such as compare-and-swap and fetch-and-add, as well as more complex operations like an *indexed Put*, where the address of a Put operation is specified by a pointer at the destination process, and the pointer is incremented after the Put completes. Such operations can be used to efficiently implement remote queues on shared memory architectures.

We are also investigating using CCS to support a hybrid UPC/MPI programming model. In such a hybrid programming environment, a process can perform both UPC and MPI operations. By porting both the Berkeley UPC runtime and MPICH2 over CCS, CCS would perform the communication operations for both paradigms, allowing both paradigms to benefit from CCS's high-performance implementation.

In remote-memory model and GAS language middleware, when a process accesses a remote portion of a shared object distributed across different processes, the virtual address of that portion at the remote process needs to be computed. This operation can be simplified by allocating shared memory regions *symmetrically* across all processes; that is, the address of the local portion of the shared object is the same at each process. This also improves the scalability of the operation because less information needs to be kept for each remote memory region. We intend to address this issue, perhaps by including a special symmetric allocation function.

Acknowledgments

We thank Rusty Lusk, Rajeev Thakur, Rob Ross, Brian Toonen, and Guillaume Mercier for their valuable comments and suggestions while we were designing and implementing CCS.

References

1. Buntinas, D., Gropp, W.: Understanding the requirements imposed by programming model middleware on a common communication subsystem. Technical Report ANL/MCS-TM-284, Argonne National Laboratory (2005)
2. Argonne National Laboratory: MPICH2. (http://www.mcs.anl.gov/mpi/mpich2)
3. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
4. Nieplocha, J., Harrison, R.J., Littlefield, R.L.: Global Arrays: A portable shared memory programming model for distributed memory computers. In: Supercomputing 94. (1994)
5. Pacific Northwest National Laboratory: Global arrays toolkit. (http://www.emsl.pnl.gov/docs/global/ga.html)
6. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, IDA, Bowie, MD (1999)
7. Lawrence Berkeley National Laboratory and University of California, Berkeley: Berkeley UPC runtime. (http://upc.lbl.gov)
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. (1997)
9. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: A mechanism for integrated communication and computation. In: Proceedings of the 19th International Symposium on Computer Architecture. (1992) 256–266
10. InfiniBand Trade Association: (InfiniBand Architecture Specification) http://www.infinibandta.com.
11. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99 (1999)
12. Bonachea, D.: GASNet specification, v1.1. Technical Report CSD-02-1207, University of California, Berkeley (2002)
13. International Business Machines: RSCT for AIX 5L LAPI Programming Guide. Second edn. (2004) SA22-7936-01.
14. Brightwell, R., Riesen, R., Lawry, B., Maccabe, A.B.: Portals 3.0: Protocol building blocks for low overhead communication. In: Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC). (2002)
15. Bell, C., Bonachea, D.: A new DMA registration strategy for pinning-based high performance networks. In: Workshop on Communication Architecture for Clusters (CAC03) of IPDPS'03. (2003)
16. Ross, R., Miller, N., Gropp, W.D.: Implementing fast and reusable datatype processing. In Dongarra, J., Laforenza, D., Orlando, S., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number LNCS2840 in Lecture Notes in Computer Science, Springer Verlag (2003) 404–413
17. Banikazemi, M., Govindaraju, R.K., Blackmore, R., Panda, D.K.: Implementing efficient MPI on LAPI for IBM RS/6000 SP systems: Experiences and performance evaluation. In: Proceedings of the 13th International Parallel Processing Symposium. (1999) 183–190
18. Myricom: The GM-2 message passing system – The reference guide to the GM-2 API. (http://www.myri.com/scs/GM-2/doc/refman.pdf)
19. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J., Su, W.: Myrinet - A gigabit per second local area network. In: IEEE Micro. (1995) 29–36