A Lightweight Idempotent Messaging Protocol for Faulty Networks

Jeremy Brown
A.I. Laboratory, M.I.T.
200 Technology Square
Cambridge, MA, 02141
(617) 253-4961
jhbrown@ai.mit.edu

J.P. Grossman
A.I. Laboratory, M.I.T.
200 Technology Square
Cambridge, MA, 02141
(617) 253-5814
jpg@ai.mit.edu

Tom Knight
A.I. Laboratory, M.I.T.
200 Technology Square
Cambridge, MA, 02141
(617) 253-7807
tk@ai.mit.edu

ABSTRACT
As parallel machines scale to one million nodes and beyond, it becomes increasingly difficult to build a reliable network that is able to guarantee packet delivery. Eventually large systems will need to employ fault-tolerant messaging protocols that afford correct execution in the presence of a lossy network. In this paper we present a lightweight protocol that preserves message idempotence and is easy to implement in hardware. We identify the requirements for a correct implementation of the protocol. Experiments are performed in simulation to determine implementation parameters that optimize performance. We find that an aggressive implementation on a fat tree network results in a slowdown of less than 2x compared to buffered wormhole routing on a fault-free network.

Categories and Subject Descriptors
C.2.2 [**Computer-Communication Networks**]: Network Protocols – *Protocol Architecture*; C.4 [**Performance of Systems**]: Fault tolerance

General Terms
Performance, Design, Reliability.

Keywords
idempotence, source-reliable messaging, block-structured traces.

1. INTRODUCTION
In large parallel machines, the implementation of the network has a first order effect on the performance characteristics of the system. Both the network topology and the messaging protocol must be carefully chosen to suit the needs of the architecture and its target applications. One of the first decisions that designers must face is whether the network or the processing nodes should be responsible for guaranteeing the successful delivery of a message.

If it is the network's responsibility, then packets injected into the network are precious and must not be corrupted or lost under any circumstances. Network nodes must contain adequate storage to buffer packets during congestion, and some strategy is required to prevent or recover from deadlock. The mechanical design of the network must afford an extremely low failure rate, as a single bad component or connection can result in system failure. Many fault-tolerant routing strategies alleviate this problem somewhat by allowing the system to tolerate static detectable faults at the cost of increased network complexity and often reduced performance. Dynamic or undetected faults, on the other hand, remain a challenge, although techniques have been described to handle the dynamic failure of a *single* link or component [11, 9, 15].

If, on the other hand, responsibility for message delivery is placed on the processing nodes, network design is simplified enormously. Packets may be dropped if the network becomes congested. Components are allowed to fail arbitrarily, and may even be repaired online so long as at least one routing path always exists between each pair of nodes. Simpler control logic allows the network to be clocked at a much higher speed than would otherwise be possible [3].

The cost, of course, is a more complicated messaging protocol which requires additional logic and storage at each node, and reduces the performance of the system. Thus, with few nodes (hundreds or thousands), it is likely a good tradeoff to place extra design effort into the network and reap the performance benefits of guaranteed packet delivery. However, as the scale of the machine increases to hundreds of thousands or even millions [19] of nodes and the number of discrete network components is similarly increased, it becomes extremely difficult to prevent electrical or mechanical failures from corrupting packets within the network. There is therefore a growing motivation to accept the possibility of network failure and to develop efficient end-to-end messaging protocols.

Any fault-tolerant messaging protocol must have the following two properties:

delivery:	All messages must be successfully delivered at least once.
idempotence:	Only one action must be taken in response to a given message even if duplicates are received.

Additionally, for a protocol to be scalable to large systems, it should exhibit these properties without storing global information at each node (e.g. sequence numbers for packets received from every other node). In light of this restriction, the idempotence property becomes more of a challenge.

In this paper we introduce a lightweight fault-tolerant idempotent messaging protocol that is easy to implement in hardware and does not require global information to be stored at each node. Each communication is broken down into three packets: the *message*, sent from sender to receiver, the *acknowledgement*, sent from receiver to sender to indicate message reception, and the *confirmation*, sent from sender to receiver to indicate that the message will not be re-sent. It will be seen that this three-part messaging arises naturally and we will show that it is not possible to reduce the number of messages without storing global information at each node.

The next section develops the idempotent messaging protocol in detail and describes the actual hardware requirements. In Section 3 we present the simulation environment used in our experiments. The results obtained from simulating several micro-benchmarks are analyzed in Section 4, where we attempt to determine implementation parameters that optimize performance, and we compare the performance of the protocol to that of buffered wormhole routing on a fault-free network. In Section 5 we discuss previous work and some existing fault-tolerant messaging protocols. We offer our conclusions in Section 6.

2. IDEMPOTENT MESSAGING

The following sections develop the idempotent messaging protocol and outline its hardware requirements. For the most part the protocol arises fairly naturally from the *delivery* and *idempotence* requirements as well as the restriction that global information may not be stored at each node. There are some subtleties, however, that must be addressed in order to ensure correctness. We begin with the assumption that the network does not reorder packets; in Section 2.3 we will see how this restriction can be relaxed.

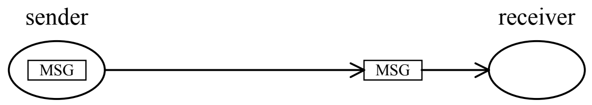
2.1 Basic Requirements

The message-acknowledge pair is fundamental to any end-to-end messaging protocol. The sender has no way of knowing whether or not a message was successfully delivered, so it must remember and periodically re-send the message until an acknowledgement (ACK) is received at which point it can forget the message.

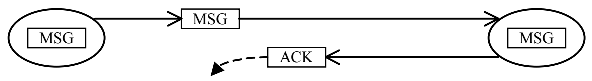
Because a message can be sent (and therefore received) multiple times, the receiver must somehow remember that it has already acted on a given message in order to preserve message idempotence. One approach, used in the TCP protocol [24], is to sequentially generate packet numbers for every sender-receiver pair; each node then remembers the last packet number that it received from every other node. This approach is feasible with thousands of nodes, but the memory requirements are likely to be prohibitive in machines with millions of nodes.

Without maintaining this type of global information at each node, the only way to ensure message idempotence is to remember individual messages that have been received. To ensure correctness, each message must be remembered until a guarantee can be made that no more duplicates will be received. This, however, depends on a remote event, specifically the successful delivery of an ACK to the sender. Only the sender knows when no more copies of the message will be sent, and so we require a third confirmation (CONF) packet to communicate this information to the receiver.

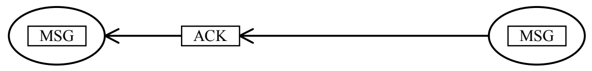
We thus have our three-part idempotent messaging protocol. The sender periodically sends a message (MSG) until an ACK is received, at which point it can drop the message. Once a message is received, the receiver ignores duplicates and periodically sends back an ACK until a CONF is received, at which point it can forget about the message. Finally, each time that a sender receives an ACK it responds with a CONF to indicate that the message will not be resent. This is illustrated in Figure 1, which shows how the protocol is able to deal with arbitrary packets being lost.



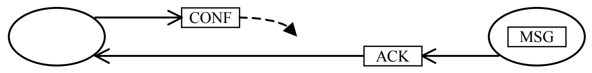
(a) Sender sends MSG and remembers it



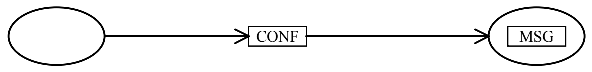
(b) Receiver remembers MSG and sends ACK. Network drops ACK. Sender re-sends MSG.



(c) Receiver ignores duplicate message and re-sends ACK



(d) Sender forgets MSG and sends CONF. Network drops CONF. Receiver sends third ACK.



(e) Sender sends CONF in response to ACK

(f) Receiver receives CONF and forgets MSG

Figure 1: Idempotent messaging example

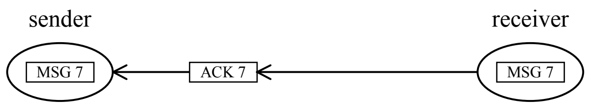
2.2 Message Identification

Each message must be assigned an identifier (ID) that can be placed in the ACK and CONF packets relating to that message. On the sending node the ID is sufficient to identify the message; on the receiving node the message is uniquely identified by the pair (source node ID, message ID). Figure 2 shows the structure of an ACK/CONF packet.

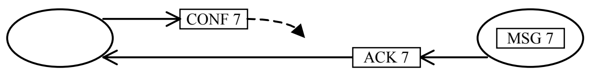
header	source node ID	message ID

Figure 2: ACK/CONF packet structure

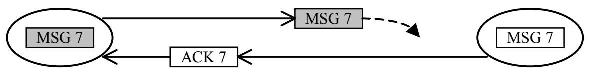
A header field is present in all packets and contains the packet type and routing information. The source node ID field identifies the node which sent the packet; for a CONF this is combined with the message ID field at the receiving node to uniquely identify the message, and for an ACK it provides the destination for the CONF response (note that this information *must* be stored in the ACK and cannot simply be remembered with the original message



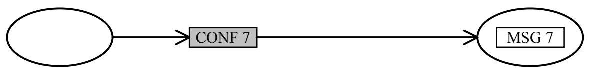
(a) Receiver sends ACK to message with ID = 7



(b) Sender forgets MSG and sends CONF. Network drops CONF. Receiver re-sends ACK.



(c) While ACK is in transit, sender re-uses ID 7 for a new message which is dropped by the network.



(d) Sender receives ACK and forgets the new message, thinking it has been received.

Figure 3: Failure resulting from message ID reuse

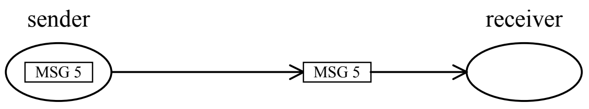
since the message is discarded when the first ACK is received, but multiple ACKs may be received).

The ACK and CONF packets represent the overhead of the idempotent messaging protocol, and as such it is desirable to make them as small as possible. It is tempting to try to use short (say 4-8 bit) message IDs and simply ensure that, on a given sending node, no two active messages have the same ID. Unfortunately, this approach fails because a message is "active" until the CONF is received, and there is no way for the sending node to know when this occurs (short of adding a fourth message to the protocol). Figure 3 shows how a message can be erroneously forgotten if message ID's are reused too quickly.

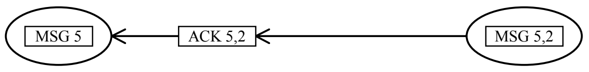
It is therefore necessary to use long message ID's so that there is a sufficiently long period between ID reuse. It is difficult to quantify "sufficiently long" since a message can, in theory, be active for an arbitrarily long time if the network continually drops its CONF packets. One possible strategy is to use 54 bit IDs so that an ACK fits into 96 bits with up to one million nodes, then drain the network by suppressing new messages once every month of operation.

The next temptation is to eliminate the source node ID field and shorten the message ID field in CONF packets only. This can be achieved by assigning to messages short secondary ID's on the receiving node so that CONF packets consist of only a header field and this secondary ID (the source node ID is no longer necessary since the secondary ID's are generated by the receiving node). Regrettably, this also fails when secondary ID's are reused prematurely. Figure 4 shows how a message can lose its idempotence when this occurs.

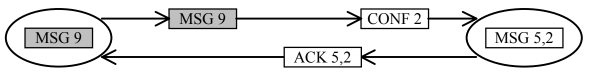
Thus, while one would like to be able to use short message IDs to reduce the size of ACK and CONF packets, we see that to do so is to sacrifice correctness.



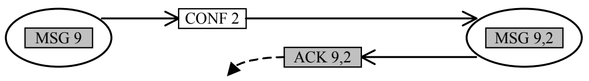
(a) Sender sends message with ID = 5



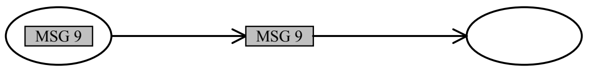
(b) Receiver assigns secondary ID = 2 and responds with ACK containing both primary and secondary ID



(c) Sender forgets MSG, sends CONF and a new MSG. Meanwhile, receiver re-sends ACK.



(d) Receiver forgets MSG 5, reuses secondary ID 2 for MSG 9, and sends ACK which is dropped by the network. Sender responds to second ACK with CONF.



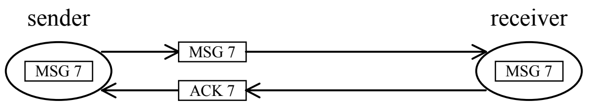
(e) Receiver receives CONF and forgets MSG 9, thinking that it will not be resent. Meanwhile, sender *does* resend MSG 9, and idempotence is lost.

Figure 4: Failure resulting from secondary ID reuse

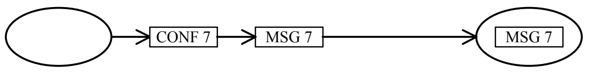
2.3 Out of Order Messages

The assumption that no more duplicate messages will be delivered once a CONF has been received is true only if packets sent from one node to another are received in the order that they were sent. If the network is permitted to reorder packets then the messaging protocol can fail as shown in Figure 5.

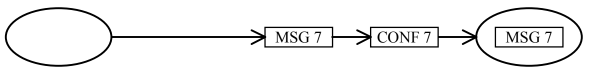
This problem can be fixed as long as the amount by which two



(a) Sender resends message just before receiving ACK



(b) Sender forgets MSG and sends CONF.



(c) Network reorders MSG and CONF. Receiver forgets MSG when it receives CONF, then idempotence is lost when duplicate MSG is received.

Figure 5: Failure resulting from packet reordering

packets can slip relative to one another is bounded. Suppose packets A, B are sent from one node to another in that order; let T be the maximum time in cycles that packet B can be received before packet A. We modify the protocol by having the receiver remember a message for T cycles after the CONF is received. Since any duplicate message would have been sent before the CONF, by choice of T it is safe to forget the message after T cycles have elapsed.

A simple way to ensure that the bound T exists is to place an upper bound on the amount of time that a packet may spend in the network. This can be accomplished either by assigning packets a time to live as in TCP [24], or by limiting the number of cycles that a packet may be buffered by a single network node before it is dropped. The latter approach is simpler as it does not require transmitting a time to live with each packet; it can be used in any network in which the length of the path taken by a packet is bounded.

2.4 Hardware Requirements

In addition to the control logic needed to implement the protocol, the primary hardware requirements are two content addressable memories (CAMs) used for remembering messages. The first of these remembers messages sent, stores {message ID, message index} on each line, and is addressed by message ID. "message index" locates the actual message and is used to free resources when an ACK is received. The processor is prohibited from generating new messages if this send table fills, and must stall if it attempts to do so until an entry becomes available. The second CAM remembers messages received, stores {source node ID, message ID} on each line and is addressed by (source node ID, message ID). No additional information is required in this CAM since the receiver simply needs to know whether or not a particular message has already been received. If this table is full, new messages received from the network are dropped.

3. SIMULATION ENVIRONMENT

Our evaluation of the idempotent messaging protocol was conducted using a trace-driven network simulator. In this section we describe the machine model that was used, the format of the traces and how they were obtained.

3.1 Hardware Model

Our hardware model is a distributed shared memory machine with explicitly split-phase memory operations. Memory consistency is enforced in software using a *wait* instruction which causes a thread to wait for all outstanding memory operations to complete before continuing. We do not model caching of remote data (which does not affect our results as all micro-benchmarks explicitly migrate data to where it is needed). Pointers contain *node* and *offset* fields; distributed objects are implemented by allocating the same range of offsets on each node.

We assume that the hardware supports efficient multithreading. Processor nodes are multithreaded, and new threads may be created with a single *fork* instruction. This instruction specifies a starting address for the new thread, the node on which the new thread should run, and a set of registers which should be copied from parent to child.

Inter-thread synchronization is register-based. A thread can create a *join capability*, a special pointer which allows other threads to write directly to one of its registers using a hardware *join* instruction. Presence bits associated with each register cause a thread to stall when it attempts to read a register being used for synchronization; the thread will resume once the corresponding join is performed and the data is available.

3.2 Block Structured Traces

Typically, the input to a trace-driven simulator is simply a set of network messages where each message specifies a source node, a destination node, the size of the message, and the time at which the message should be sent. These traces may be obtained by instrumenting actual parallel programs running on multiple real or simulated processor nodes.

There are two problems with this straightforward approach. First, in an actual program the time at which a given message is sent generally depends on the time that one or more previous messages were received. It is therefore inaccurate to specify this time a priori in a trace. Second, a large parallel computer may not be readily available, and the number of threads required to run a parallel program on thousands of simulated nodes can easily overwhelm the operating system.

We address the first problem by organizing the trace into **blocks** of timed messages. Each block represents a portion of a thread in the parallel program which can execute from start to finish without waiting for any network messages. When a block is activated, each of its messages is scheduled to be sent at a specified number of cycles in the future. Each message optionally specifies a target block to signal when the message is successfully delivered; a block is activated when it has been signaled by all messages having that block as a target. This block-structured trace captures the dependency graph of messages within an application, and allows the simulation to more accurately reflect the pattern of messages that would arise from running the parallel program with a given network configuration.

Block-structured traces are a similar to *intrinsic traces* [17], used in trace-driven memory simulators to model programs whose address traces depend on the execution environment. It has been observed that trace-driven parallel program simulations can produce unreliable results if the traces are of timing-dependent code [17, 16]; our micro-benchmarks and synchronization mechanisms were therefore chosen to ensure deterministic program execution.

3.3 Obtaining the Traces

The second problem – the difficulty of simulating thousands of nodes on a single processor – is addressed by our method of obtaining traces. We provide a small library of routines that implement the hardware model described in Section 3.1; these functions are listed in Table 1. The routines are instrumented to transparently manage blocks, messages, and the passage of time. Most importantly, they are designed to allow the program to run as a

Table 1: Simulation library functions

Function	Description
Load	Load data from a (possibly remote) location
Store	Store data to a (possibly remote) location
Wait	Wait for all outstanding stores to complete
Fork	Start a new thread of execution
Join	Write data to another thread's registers
Sync	Register synchronization: wait for a join

```
void Fork (_thread t,   /* thread entry point */
           int node,    /* destination node */
           ...);        /* other arguments */

int ComputeSum (Pointer data)
{
    JCap *j = new JCap;
    Fork(SumThread, 0, numNodes, data, j);
    return Sync(j);
}

void SumThread (int cNodes, Pointer data, JCap *j)
{
    if (cNodes > 1)
    {
        int n = cNodes / 2;
        JCap *j1 = new JCap;
        JCap *j2 = new JCap;
        Fork(SumThread, node, n, data, j1);
        Fork(SumThread, node+n, cNodes-n, data, j2);
        Join(j, Sync(j1) + Sync(j2));
    }
    else
    {
        data.node = node;
        Join(j, Load(data));
    }
}
```

Figure 6: Sample program to compute the sum of a distributed object with one word on each node. *node* and *numNodes* are global variables.

single thread. This is primarily accomplished by implementing the Fork routine using a function call rather than actually creating a new thread. Figure 6 gives a very simple program written with this library.

As an example of how the library routines are implemented, Figure 7 gives simplified code for Load. Two global variables, *block* and *node*, are managed by the library routines and contain, respectively, a pointer to the block corresponding to the current thread of execution, and the processing node on which the thread is running. If the load is from local memory, the routine simply adds time to the current block. If the load references a remote memory location, a message is added to *block* representing the load request. The message's destination is the remote node, and it signals a new block *ref* which contains a single message representing the reply to the load request. The destination of the reply message is *node*, and it signals a second new block *next* which represents the continuation of the current thread once the value of the load has been received (it is assumed that the current thread must wait for this value – we are not taking prefetching into account). Finally, the current block is set to *next*.

The actual Load routine is slightly more complicated as it also checks for address conflicts with outstanding stores. The Wait routine is provided to enforce memory consistency by explicitly waiting for all stores to complete before execution continues.

While the library routines automatically manage the passage of time for the parallel primitives that they implement, it is the programmer's job to manage the passage of time for normal computation. A macro is provided for adding time to the current block. The programmer is responsible for making use of this macro and providing a reasonable estimate of the number of cycles required to perform a given computation.

3.4 Synchronization

In the simulation environment, register-based synchronization is accomplished using the Sync and Join library routines. There are no actual registers in the simulation, so Join is implemented by storing a word of data in the join capability data structure (and adding a message to the current block); Sync retrieves the word from the data structure (and creates a new block).

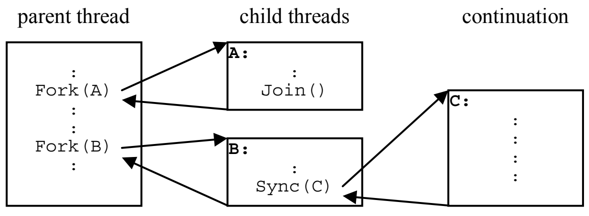
Because the simulation is run as a single thread, the straightforward implementation of Sync will only work if the data is already available, i.e. if the corresponding Join has already been called. If all synchronization is from child to parent then this will always be the case because implementing Fork using a function call causes the "threads" to run to completion in a depth-first manner. Figure 6 gives an example of child to parent synchronization. While each Fork conceptually creates a new thread, the single-threaded implementation simply calls SumThread as a subroutine and then returns, so Join will already have been called by the time the parent thread calls the corresponding Sync.

To allow for more complicated synchronization wherein Sync may be called before the corresponding Join, a version of Sync is provided in which the programmer explicitly provides a continuation. If the data is ready when Sync is called, then the continuation is invoked immediately. Otherwise the continuation is stored in the join capability data structure and invoked when the corresponding Join is called (Figure 8). This sacrifices some of the

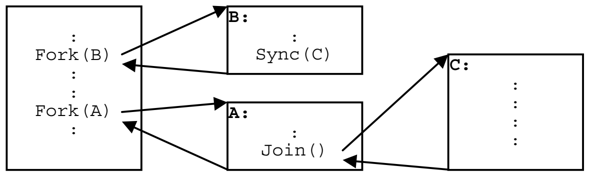
```
void Block::AddMessage
     (int size,        /* size of message */
      int dst,         /* destination node */
      Block *target);  /* block to signal */

Word Load (Pointer p)
{
    if (p.node == node)
        block->time += LOCAL_REF_TIME;
    else
    {
        Block *ref = new Block;
        Block *next = new Block;
        ref->node = p.node;
        next->node = node;
        block->AddMessage(LOAD_SIZE, p.node, ref);
        ref->AddMessage(REPLY_SIZE, node, next);
        block = next;
    }
    return memory[p];
}
```

Figure 7: Load routine (simplified). *block* and *node* are global variables.



(a) Join called before Sync; continuation invoked by Sync



(b) Sync called before Join; continuation invoked by Join

Figure 8: Control flow of single-threaded simulation with user-supplied continuations.

transparency of the simulation environment in order to retain the benefits of being able to run the simulation using a single thread.

4. EXPERIMENTAL RESULTS

The purpose of our simulations is to explore the parameter space of the idempotent messaging protocol and to compare its performance to that of buffered wormhole routing on a fault-free network. The two parameters of interest are the strategy used for packet retransmission and the size of the message tables. In the following sections we present the micro-benchmarks and network models used in simulation, and we give the results of our experiments.

4.1 Micro-Benchmarks

Four micro-benchmarks were chosen to provide a range of network usage patterns. Each one was coded as described in Section 3. The four resulting block-structured traces were used to drive our simulations. The micro-benchmarks are as follows:

add: Parallel prefix addition on 4096 nodes with one word per node. Light network usage. Network is used for synchronization and thread creation.

reverse: Reverse the data of a 16K entry vector distributed across 1024 nodes. Very heavy network usage with almost all messages crossing the bisection.

quicksort: Parallel quicksort of a 32K entry random vector on 1024 nodes. Medium, irregular network usage (lighter than *reverse* or *nbody* due to a higher computation:communcation ratio).

nbody: N-body simulation on 256 nodes with one body per node. Computation is structured for $\sqrt{N}$ communication by conceptually organizing the nodes in a square array and broadcasting the location of each body to all nodes in the same row and column. Heavy but regular network usage.

4.2 Network Model

In an attempt to ensure that our results are independent of the network topology, three different topologies are used in all simulations: a 2D grid, a 3D grid, and a fat tree where each parent node connects to four child nodes and bandwidth doubles with each step up the tree. For the grid networks dimension-ordered routing is preferred, but any productive channel may be used to route the packet. In all cases wormhole routing is used, with packet heads advancing one step per cycle. If a packet cannot be advanced due to congestion, it is dropped. This, combined with the fact that the distance between a given pair of nodes is fixed, ensures that the network will not reorder packets. Flits are 32 bits wide, and each packet ends with a checksum flit. ACK and CONF packets are four flits each (including the checksum).

4.3 Packet Retransmission

In order for the idempotent messaging protocol to function correctly, it is necessary to periodically retransmit MSG and ACK packets. When such a packet is sent, it should be scheduled for retransmission at

$$size + 2 \times distance + \text{ACK\_SIZE} + backoff$$

cycles in the future. The first three terms in this sum represent the amount of time it takes to receive an ACK/CONF packet if the receiving node is able to reply immediately and if neither packet is dropped by the network. The *backoff* term is a function of the number of transmit attempts for the packet (n), and represents the strategy being used to manage network congestion.

Four backoff terms were considered: constant (C), linear (Cn), quadratic (Cn^2) and exponential ($C \cdot 2^n$). We do not present results for constant or exponential backoff as their performance was unacceptable. A constant backoff is intuitively bad as it makes no attempt to manage congestion, and indeed in simulation it often caused livelock when the network became congested. Exponential backoff was found to be overkill; in a congested network packets were often rescheduled with large delays and as a result performance suffered.

Figure 9 shows plots of execution time for all four micro-benchmarks on all three topologies with both linear and quadratic backoff as the retransmission constant C is varied from 1 to 32.

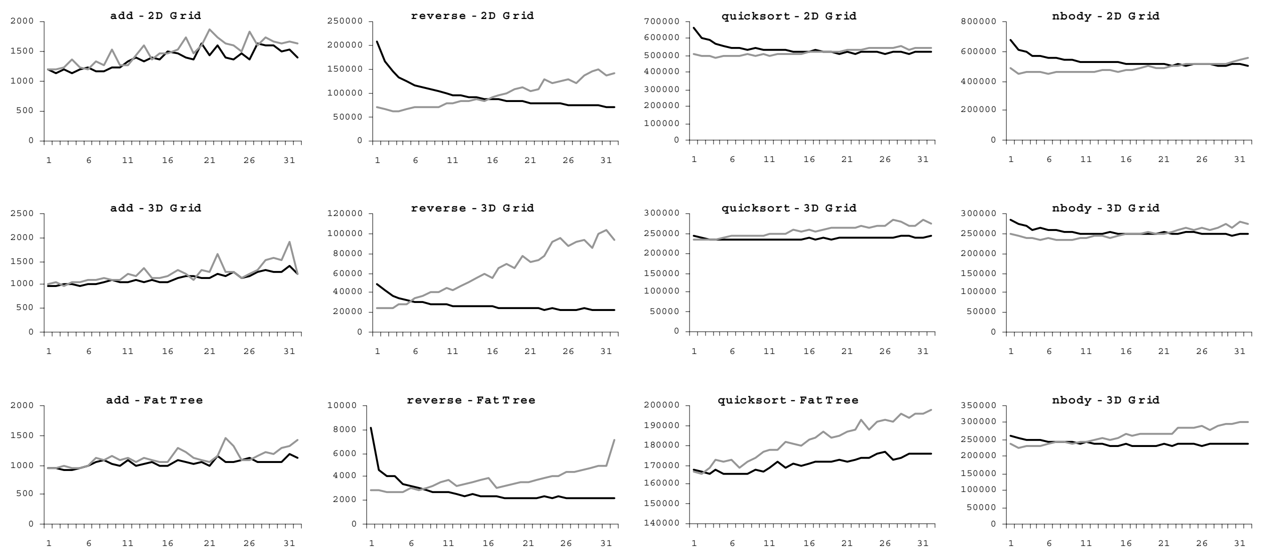


Figure 9: Execution time in cycles vs. retransmission constant *C* for linear (——) and quadratic (——) backoff

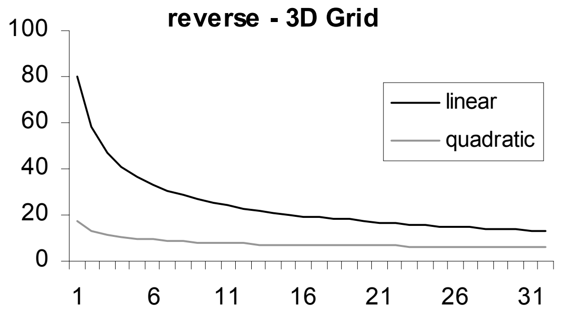


Figure 10: Average message retransmission vs. retransmission constant *C* for *reverse* on a 3D Grid.

We see that quadratic backoff performs well with small *C*, whereas linear backoff requires large *C* unless the network is lightly loaded (compare the linear backoff curves for *quicksort* on the three topologies; 2D Grid has the smallest capacity and Fat Tree has the largest capacity). Intuitively this indicates that linear backoff is not quite enough; further evidence supporting this intuition is provided by graphing average message retransmission against *C* for linear and quadratic backoff. Figure 10 shows this graph for reverse on a 3D Grid; the corresponding graphs for other micro-benchmarks and topologies are nearly identical with varying vertical scales. We see that the number of message transmissions is much larger when linear backoff is used, indicating that the available network bandwidth is not being used efficiently.

It is difficult to determine from inspecting the graphs which retransmission strategy is "best". Resorting to numerical analysis, we asked the question of which strategies provided closest-to-optimal performance in the worst and average cases (where "optimal" refers to the best observed performance for a given benchmark/topology combination). We found quadratic backoff with *C* = 3 to be superior under both metrics, performing within 28% of optimal in the worst case and within 7% of optimal in the average case. We therefore use these settings for the remainder of the simulations.

4.4 Table Size

The next important implementation parameter is the size of the tables used to remember messages that have been sent/received. There is a tradeoff between performance and implementation cost since the tables require expensive content addressable memory, but if a table fills up it will temporarily prevent new messages from being sent or received. Note that having a receive table fill up is much worse than having a send table fill up as it will cause messages to be dropped *after* they have traversed the network, which wastes network bandwidth. Accordingly, in the simulations we use receive tables which are twice as large as the send tables.

In Figure 11 execution time is graphed for all micro-benchmarks and topologies as send table size is varied from 2^0 to 2^{10}. In most cases execution time quickly drops to a minimum, and we can achieve near-optimal performance with as few as 8 send and 16 receive table entries. The exception is *reverse*, where execution time actually increases with larger table sizes. This is due to the increased network congestion that results when nodes are able to send more messages.

We note for the sake of completeness that when the send and receive tables are the same size the graphs rapidly approach the same minima, but the initial slope is much steeper and performance is 20-30% worse for tables sizes of 1, 2 and 4.

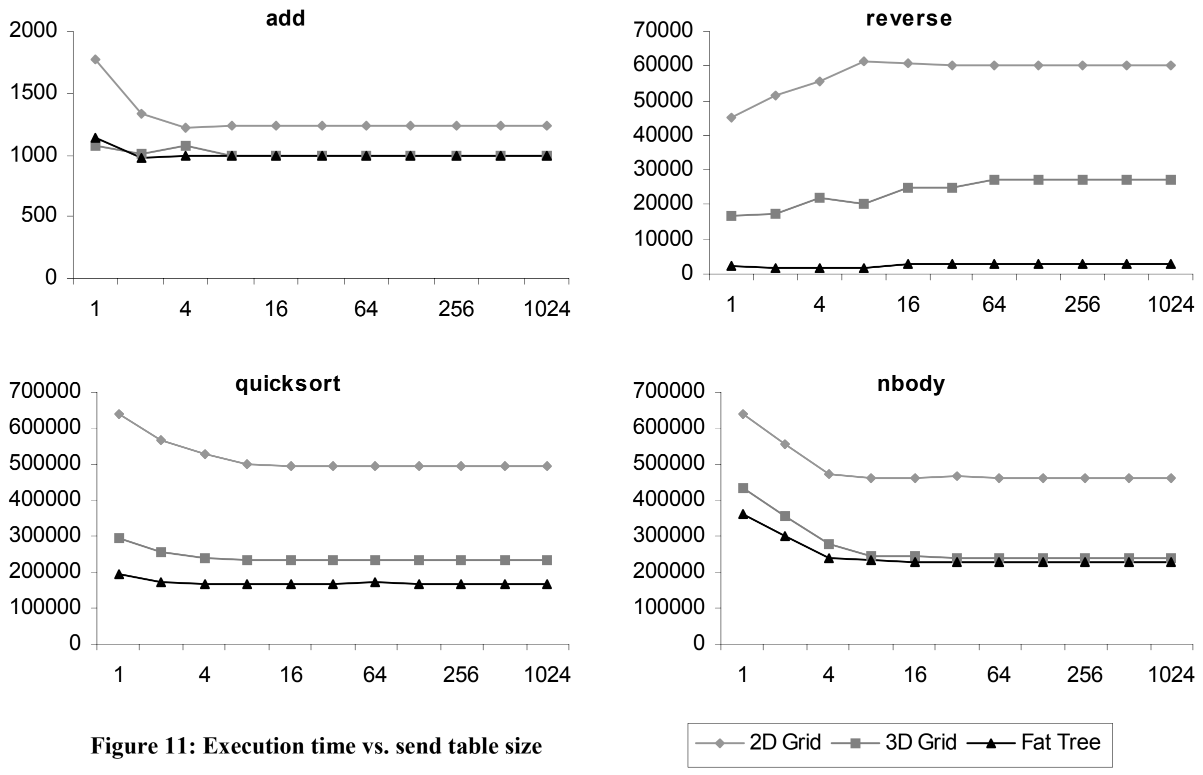


Figure 11: Execution time vs. send table size

Table 2a: Slowdown of idempotent messaging protocol compared to buffered wormhole routing on a perfect network

topology:	**2D Grid**			**3D Grid**			**Fat Tree**		
	buffered (cycles)	idempotent (cycles)	slow-down	buffered (cycles)	idempotent (cycles)	slow-down	buffered (cycles)	idempotent (cycles)	slow-down
add	814	1229	**1.51**	645	986	**1.53**	634	991	**1.56**
reverse	8198	60548	**7.39**	3333	24799	**7.44**	717	2701	**3.77**
quicksort	195356	496306	**2.54**	124230	232580	**1.87**	96745	167438	**1.73**
nbody	119528	463741	**3.88**	79493	241771	**3.04**	97720	229111	**2.34**

Table 2b: Slowdown when 40 bit flits and a 60% faster clock are used for the idempotent messaging protocol

topology:	**2D Grid**			**3D Grid**			**Fat Tree**		
	buffered (cycles)	idempotent (cycles)	slow-down	buffered (cycles)	idempotent (cycles)	slow-down	buffered (cycles)	idempotent (cycles)	slow-down
add	814	848	**1.04**	645	709	**1.10**	634	637	**1.00**
reverse	8198	30053	**3.67**	3333	13931	**4.18**	717	1421	**1.98**
quicksort	195356	285565	**1.46**	124230	147229	**1.19**	96745	113460	**1.17**
nbody	119528	265500	**2.22**	79493	137627	**1.73**	97720	131092	**1.34**

4.5 Performance Comparison

As stated in the introduction, for small systems it is probably worth constructing a reliable network, whereas for extremely large systems it is almost certainly necessary to implement a fault-tolerant messaging protocol. In between these extremes, however, it may be difficult to determine which approach is more appropriate, and it becomes useful to know the performance impact of an idempotent messaging protocol.

To compare the two approaches, we simulated a perfect network with buffered wormhole routing. Each network link contains enough storage to buffer an entire packet, so when the network is congested packets are stored rather than dropped. We assume that nodes always have enough available resources to accept incoming packets. For the grid networks, strict dimension-ordered routing is used to avoid deadlocks [7]. Because the network is assumed to be perfect, the checksum flit is omitted. The idempotent messaging protocol was simulated using 16-entry send tables and 32-entry receive tables.

The results of the comparison are shown in Table 2a. We see that the actual slowdown, which ranges from as little as 1.53 to as much as 7.44, depends on both the application and the network topology. In general, a more congested network leads to greater slowdowns. Note that in our simulations we are assuming that the flit size and cycle times of the two networks are the same. In practice this would likely not be the case for three reasons. First, the control logic of the lossy network is much simpler than that of the lossless network; as a result it will be possible to clock the lossy network at a higher speed [3]. Second, with a fault-tolerant messaging protocol and enough checksum bits it is possible to boost the clock speed even further since one does not need to worry about introducing the occasional signaling error so long as it can be detected. Finally, in the lossless network a number of bits would need to be added to each flit for the purposes of error correction. For example, at least 10 extra bits are required in order to perform double error correction on 32 bit flits. Thus, if both networks are built using the same number of physical bits per connection, the lossy network will have larger flits.

If we again compare the two approaches under the assumptions that the fault-tolerant network has flits which are 25% larger and a clock that runs 60% faster, we obtain the results shown in Table 2b. In this case the slowdowns are far less severe, and are less than 2x across all micro-benchmarks on the fat tree network.

5. RELATED WORK

The protocol described in this paper was first reported in [28]; it was implemented as part of a faulty network simulation in [29].

The practice of discarding packets is common among WAN networking technologies such as Ethernet [21] and ATM [27]; end-to-end protocols such as TCP [24] are required to ensure reliable message delivery over these networks. However, WAN-oriented protocols generally require total table storage proportional to N^2 for N inter-communicating nodes [11, 8], and are therefore poorly suited to large distributed shared-memory machines.

Only a few parallel architectures feature networks which may discard packets; among these exceptional cases are the BBN Butterfly [25], the BBN Monarch [26], and the Metro router architecture [4]. Each of these implements a circuit-switched network which discards packets in response to collisions or network faults.

While specific types of operations may be transformed into idempotent forms for repeated transmission over unreliable networks [13], no general mechanism providing lightweight end-to-end idempotence has previously been reported. As a result, most parallel architectures have implemented non-discarding networks which take responsibility for message delivery [18, 22, 6]. Such networks can only tolerate limited types of network failure.

Many parallel networks can be configured to route around static faults, e.g. the Cray T3D [5]. By contrast, very few networks are able to handle dynamic faults, but examples do exist: the SGI Spider router [15] can detect and disable failing links at runtime, and the Reliable Router chip [9] can tolerate the dynamic failure of individual links or components. The RR chip uses a link-level protocol, the Unique-Token Protocol (UTP) [11], which has two interesting similarities to our lightweight idempotence protocol: first, as with our protocol, the UTP requires receiving nodes to detect and ignore duplicate copies of a message; second, the UTP uses a four-step handshake between router chips which is rather similar to our protocol's three-step end-to-end handshake. Neither the Reliable Router nor the Spider router are able to handle multiple dynamic failures.

Non-discarding networks often suffer from congestion problems which can fill the network with undelivered messages [12, 23, 20, 6]. Both link-level [1] and end-to-end [2, 14] protocols have been proposed which improve system performance by limiting message injection rates; the end-to-end flow control protocol uses a two-message handshake identical to the first two messages of our three-message handshake. By contrast, discarding networks do not experience "traffic jams" in the network, and with an appropriate backoff strategy may even outperform non-discarding networks under certain high-load conditions [25, 10].

Non-discarding networks rely on topological constraints and virtual channels to avoid deadlock [7, 8]. However, virtual channel implementation has a fairly severe impact on network component latency and clock speed [3]. Discarding networks are generally immune to deadlock; livelock can avoided by using an appropriate backoff strategy.

6. CONCLUSION

To date there has been very little work on end-to-end fault-tolerant messaging protocols in the context of parallel machines, and with good reason: existing systems are small enough that it is possible to implement non-discarding networks. As shown in Section 4.5, this is clearly the better approach when performance is the only concern. When the number of nodes is increased by one or two orders of magnitude, however, it becomes increasingly difficult to engineer a reliable network, and so the focus must shift to the development of reliable end-to-end network protocols.

In this paper we have shown that it is possible to implement a lightweight messaging protocol that guarantees message delivery and idempotence, can tolerate arbitrary dynamic faults (so long as at least one routing path exists between every pair of nodes), and does not require the expected $O(N^2)$ resources for N nodes. The key element of the protocol is the third *confirm* packet which informs the receiver that no more duplicate messages will be sent.

While the idempotent messaging protocol is more demanding on the network – the number of packets is more than tripled, counting retransmissions – it also allows for larger flits, faster clocks, and more flexible routing. We have seen that it is reasonable to expect the performance degradation to be less than 2x in an actual implementation on a high-bandwidth network. On some systems it may be possible to further improve performance by piggybacking the response to remote memory references in the acknowledgement packet. Another potential performance enhancement is to implement a hybrid network that is able to buffer packets during congestion. Note that the combination of packet buffering and multiple routing paths implies an out-of-order network, so the protocol would need to be modified as described in Section 2.3.

We note that our discussion of retransmission backoff strategies is preliminary at best; the space of possible approaches is enormous and a more comprehensive evaluation was beyond the scope of this paper. We chose to focus on simple functions of the transmission attempt counter n. Using a lookup table, one could in fact implement arbitrary functions of n. More generally, the backoff could depend on other factors as well such as estimated measures of congestion around the source node, the destination node, or in the network as a whole.

7. ACKNOWLEDGEMENTS

Thanks to Dr. Andrew "bunnie" Huang for his help and suggestions. This work was supported by DARPA/AFOSR contract number F306029810172.

8. REFERENCES

[1] E. Baydal, P. Lopez, J. Duato, "A congestion control mechanism for wormhole networks", Proc. Ninth Euromicro Workshop on Parallel and Distributed Processing, 2001, pp. 19-26.

[2] Timothy J. Callahan, Seth Copen Goldstein, "NIFDY: A Low Overhead, High Throughput Network Interface", Proc. ISCA 1995, pp. 230-241.

[3] Andrew A. Chien, "A Cost and Speed Model for *k*-ary *n*-Cube Wormhole Routers", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 2, 1998, pp. 150-162.

[4] Frederic Chong, Henry Minsky, André DeHon, Matthew Becker, Samuel Peretz, Eran Egozy, Thomas F. Knight Jr., "METRO: A Router Architecture for High-Performance, Short-Haul Routing Networks", Proc. ISCA 1994, pp. 266-277.

[5] Cray Research, "CRAY T3D System Architecture Overview", Cray Research Inc., March 1993. 169 pp.

[6] D. Culler and J. Singh and A. Gupta, "Parallel Computer Architecture. A Hardware/Software Aproach." Morgan Kaufmann Publishers, Inc, San Francisco, 1999.

[7] William J. Dally, Charles L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks", IEEE Transactions on Computers, Vol. 36, No. 5, 1987, pp. 547-553.

[8] William J. Dally, Hiromichi Aoki, "Deadlock-free Adaptive Routing in Multicomputer Networks using Virtual Channels", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4, 1993, pp. 466-475.

[9] William J. Dally, Larry R. Dennison, David. Harris, Kinhong Kan, Thucydides Xanthopoloulos, "The Reliable Router: A Reliable and High-Performance Communication Substrate for Parallel Computers", Proc. First International Parallel Computer Routing and Communication Workshop, Seattle, WA, May 1994.

[10] Suprakash Datta, Ramesh Sitaraman, "The Performance of Simple Routing Algorithms that Drop Packets", Proc. SPAA 1997, pp. 159-169.

[11] Larry R. Dennison, "Reliable Interconnection Networks for Parallel Computers", MIT Artificial Intelligence Laboratory technical report AITR-1294, 1991, 78pp.

[12] Eric A. Brewer, Bradley C. Kuszmaul, "How to Get Good Performance from the CM5 Data Network", Proc. 1994 International Parallel Processing Symposium, Cancun, Mexico, April 1993, pp. 858-867.

[13] Ian Eslick, André DeHon, Thomas Knight Jr., "Guaranteeing Idempotence for Tightly-coupled, Fault-tolerant Networks", Proc. First International Workshop on Parallel Computer Routing and Communication, 1994, pp. 215-225.

[14] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee,

"The M-Machine Multicomputer", Proc. MICRO-28, 1995, pp. 146-156.

[15] M. Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SPIDER Chip", Proc. Hot Interconnects Symposium IV, August 1996, pp. 141-146.

[16] Stephen R. Goldschmidt, John L. Hennessy, "The Accuracy of Trace-Driven Simulations of Multiprocessors", Measurement and Modeling of Computer Systems, 1993, pp. 146-157.

[17] Mark A. Holliday, Carla Schlatter Ellis, "Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation", IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 1, 1992, pp. 97-109.

[18] Kai Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability", McGraw-Hill New York, 1993, 771 pp.

[19] F. Allen et. al., "Blue Gene: A Vision for Proten Science using a Petaflop Supercomputer", IBM Systems Journal, Vol. 40, No. 2, 2001, pp. 310-327.

[20] Vijay Karamcheti, Andrew A. Chien, "A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D", Proc. ISCA 1995, pp. 298-307.

[21] Robert Metcalfe, David Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks (Reprint)", Communications of the ACM, Vol. 26, No. 1, 1983, pp. 90-95.

[22] Lionel M. Ni, Philip K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks", IEEE Computer, Vol. 26, No. 2, 1993, pp. 62-76.

[23] Michael D. Noakes, Deboarah A. Wallach, William J. Dally, "The J-Machine Multicomputer: An Architectural Evaluation", Proc. ISCA 1993, pp. 224-235.

[24] J. Postel, "Transmission Control Protocol", RFC 793, 1981.

[25] Randall Rettberg, Robert Thomas, "Contention is no obstacle to shared-memory multiprocessing", Communications of the ACM, Vol. 29, No. 12, 1986, pp. 1202-1212.

[26] Randall Rettberg, William R. Crowther, Philip P. Carvey, Raymond S. Tomlinson, "The Monarch Parallel Processor Hardware Design", IEEE Computer, Vol. 23, No. 4, 1990, pp. 18-30.

[27] Reza Rooholamini, Vladimir Cherkassky, Mark Garver, "Finding the Right ATM Switch for the Market", IEEE Computer, Vol. 27, No. 4, 1994, pp. 16-28.

[28] Jeremy Brown, "An Idemptent Message Protocol", Project Aries Technical Memo ARIES-TM-014, available at http://www.ai.mit.edu/projects/aries/documents

[29] Bobby Woods-Corwin, "A High Speed Fault-Tolerant Interconnect Fabric for Large-Scale Multiprocessing", M.Eng Thesis, Dept. of EECS, M.I.T., May 2001.