Thread-based vs Event-based Implementation of a Group Communication Service

Shivakant Mishra and Rongguang Yang
Department of Computer Science
University of Wyoming, P.O. Box 3682
Laramie, WY 82071-3682, USA.
Email: {mishra|chong}@cs.uwyo.edu

Abstract

We evaluate two techniques to implement concurrent events in a group communication service. This evaluation is based on a comparison of the performance measured from two different implementations of a group communication service, and our experience in using the two techniques to implement group communication services in the past. Our conclusion is that an event-based implementation is preferable to a thread-based implementation of a group communication service.

1. Introduction

Most group communication services are event driven, i.e. group members invoke various functions on the occurrence of an event. In general, several events may occur concurrently at a group member implementing a group communication service. There are two common techniques to implement concurrent, event-driven software: *thread-based* and *event-based*. In the thread-based technique, a separate thread is spawned for each event type, say e_{type}, in the program. This thread waits for an event of type e_{type} to occur and takes appropriate actions on the occurrence of that event. In the event-based technique, a single-threaded event loop performs event demultiplexing and event handler dispatching in response to the occurrence of multiple events.

While both techniques have been used to handle concurrent events in group communication services [2, 7, 3, 8, 1], it is not clear which of them is more suitable. The goal of this paper is to evaluate these two techniques in implementing group communication services in terms of their effect on service performance, complexity of their implementation, and portability. This evaluation is based on an implementation of two versions (event-based and thread-based) of the Timewheel group communication service [5, 6], and our experience in using these techniques to implement group communication services in the past. The main conclusion is that an event-based technique is almost always preferable to a thread-based technique to handle concurrent events in group communication services. An event-based implementation results in better performance, simpler, more manageable, and portable code.

2. Concurrent Events

Events in a group communication service occur due to interactions between a client and a group member, interactions among group members, and various timeouts set by group members. The Timewheel group communication service [5, 6] consists of a clock synchronization protocol, a group membership protocol, and an atomic broadcast protocol. It supports nine group communication semantics: three kinds of ordering semantics—*no order*, *total ordered*, and *time ordered*, three kinds of atomicity semantics—*weak*, *strong*, and *strict*, and one termination semantic.

There are two types of interaction between a client and a group member in the Timewheel group communication service. A client can request to multicast a service state update with a given pair of atomicity and order semantics, or it can request to retrieve some information about the replicated service state.

There are two types of messages sent by group members to implement the Timewheel atomic broadcast protocol: a *proposal message* to disseminate a service state update and a *decision message* to disseminate ordinal information. There are three types of messages exchanged among members to implement the Timewheel membership protocol: a *no-decision message* to inform a failure suspicion, a *join message* to request joining a group, and a *reconfiguration message* to disseminate information about the list of live processes. Finally, the clock synchronization protocol uses a message to synchronize clocks of different members.

The Timewheel group communication service uses four types of timeouts: a *decision message timeout* used to set a time by which the next decision message is expected, a *decider timeout* used by a member(decider) to set a time by which the next decision message is to be sent, a *decider election timeout* used by the membership protocol to elect a new decider, and a *clock synchronization timeout* used by the clock synchronization protocol to set a time when the next clock synchronization message is to be sent.

3. Thread-Based Implementation

The thread-based implementation of the Timewheel group communication service uses the SGI's POSIX-compliant, pthreads library. Six threads are used in this implementation. One thread executes in a loop waiting to receive a request from a client and, on receiving a request, invokes a function to service that request. Another thread executes in a loop waiting to receive a message from the other members and, on receiving a message, invokes a function to process that message. The remaining threads implement timers: one each in the atomic broadcast and clock synchronization protocols and two in the group membership protocol. A timer is implemented as follows:

```
void timer(pthread_mutex_t *m, pthread_cond_t *c,
        struct timespec *interval)
{
  pthread_mutex_lock(m);
  pthread_cond_timedwait(c, m, interval);
  pthread_mutex_unlock(m);
}
```

The reason for using `pthread_cond_timedwait` function, instead of a simple timed-wait function to implement timers is that, in some cases, we need to unblock the timer thread before the specified time interval has passed. This can be done by `pthread_cond_signal` function call. Some of these timer threads are not always needed. For example, the second timer thread in the membership protocol is needed only when a failure occurs and it is determined that the decider role has been lost. In our implementation, these timer threads are blocked on a `pthread_cond_wait` function call, when they are not needed. They are unblocked by the `pthread_cond_signal` function call, when needed.

Progress of the Timewheel group communication service is dependent on the individual clocks of different processors being synchronized. A group member whose clock is not synchronized with the other members leaves the group (See [4] for details). So, to ensure progress, the timer thread used by the clock synchronization protocol needs to be scheduled as soon as its timer expires. We give this thread the highest priority. The other five threads are of equal priorities.

A consequence of unequal priorities is that race conditions can occur between them, even when the service is implemented on a network of uni-processor systems[1]. A running thread may get interrupted in the middle of executing a function, and since all six threads execute in the same address space, the threads scheduled later may interfere with the preempted thread. Unfortunately, there is very little parallelism in group communication services. In general, execution of a function invoked by a thread must complete before another thread can invoke a function. So, to handle race conditions, we simply enclose most functions by semaphores.

To minimize the possibility of preemption of a running thread by the scheduler, we have used the fifo scheduling policy. To ensure that each of the five equal-priority threads get a fair share of the CPU, a running thread yields its control of CPU, whenever it finishes the execution of the function invoked.

4. Event-Based Implementation

There are six types of events in the event-based implementation of the Timewheel group communication service. They correspond to the six threads in the thread-based implementation. An outline of the event loop and the processing of timeout events is shown below:

```
void mainloop()
{
  for (;;) {
    rset = readfds;
    if (to_ev_queue→queue_size() > 0)
      to_ev_queue→get_timeout(to);
    else to = NULL;
    ret = select(maxFD, &rset, NULL, NULL, to);
    if (ret == 0) process_to();
    else for (int i = minFD; i < maxFD; i++)
      if (FD_ISSET(i, &rset)) {
        ln = recvfrom(i, sm, mlen, 0, NULL, &ret);
        event_table[i]→new_event();
      }
  }
}
void process_to()
{
  int done = 0;
  to_ev_queue→first→new_event();
  to_ev_queue→dequeue();
  do {
```

[1] If the service is implemented on a network of multiprocessor systems, race conditions between threads can occur even when all threads are of equal priorities. This can happen if the thread package can schedule more than one thread concurrently at different processors.

```
        if (time_over(clk_sync_to→time)) {
            to_ev_queue→remove_event(clk_sync_to);
            clk_sync_to→event();
        }
        else if (time_over(to_ev_queue→first→time)) {
            to_ev_queue→first→new_event();
            to_ev_queue→dequeue();
        }
        else done = 1;
    } while(!done);
}
```

The event loop is implemented in a standard way using the Unix `select()` system call: the receive events are registered in the `fd_set` *readfds* using the `FD_SET` function call and the earliest timeout event is included in the `timeval` struct *to*. A queue *to_ev_queue* of timeout events is maintained that stores all timeout events registered. This queue is sorted by the time the timeout events are suppose to occur. Because timely processing of timeout events is important, a timeout event is processed before a message-receipt event.

To ensure that a clock synchronization timeout event is given priority over any other timeout event, a separate variable *clk_sync_to* is maintained to record when the next clock synchronization timeout event will occur. After processing the first timeout event, if the clock synchronization timeout event has occurred, it is processed.

5. Evaluation

5.1. Complexity of Implementation

Concurrent events raise two related programming problems that need to be addressed: race conditions and scheduling. Concurrent events can cause race conditions in a thread-based implementation, and so, appropriate synchronization primitives have to be used. This adds complexity in the code. In general, concurrent programming is hard and extra attention has to be paid to ensure correctness.

Because not all threads have equal priority in a thread-based implementation, a running thread may be preempted in the middle of a function execution. To ensure that all threads get a fair share of the CPU, they have to explicitly yield the CPU control at some logical points in their execution. Hence, the scheduling of the threads has to be controlled, to some extent, by implementors in a thread-based implementation.

Since, there is only one thread of control in an event-based implementation, the problems of race conditions or scheduling do not arise. However, in an event-based implementation, an event loop has to be implemented that schedules the processing of events in an appropriate order. Special care has to be taken in the implementation of this event loop to ensure that higher priority events, such as the timeout event in the clock synchronization protocol, are processed first.

We found that an event-based implementation resulted in a simpler and more manageable code. Using synchronization primitives correctly and controlling the scheduling of threads in the thread-based implementation was much more complicated than implementing the event loop in the event-based implementation.

5.2. Portability

A thread-based implementation needs a threads package, either implemented inside the operating system kernel or at the user level, while an event-based implementation needs a system call such as `select()` to examine an I/O descriptor set for event scheduling. Our experience in implementing group communication services has been mostly on a Unix platform. While the `select()` system call is portable among all types of Unix operating system, a threads package is not always portable. For example, we implemented the Pinwheel atomic broadcast protocols on a network of Sun IPX SPARCstations running SunOS 4.1.2 Unix operating system [3]. This implementation used the Sun's LWP thread management library that runs on SunOS 4.1.2. Unfortunately, this thread management library does not run on other Unix operating systems such as Solaris or IRIX. As a result, we had to do a lot of work to run the Pinwheel protocols on an IRIX operating system platform.

So, while an event-based implementation is fairly portable, the portability of a thread-based implementation is limited by the portability of the threads package used. In the past, when no standardized threads packages existed, this was a major problem. However, with standard threads packages such POSIX-compliant pthreads becoming more common, this portability problem with a thread-based implementation is expected to diminish.

5.3. Performance

There are four performance indices we have measured: *throughput*, average broadcast *delivery time*, average broadcast *stability time*, and average number of messages exchanged per atomic broadcast.

Throughput measured from the event-based implementation is higher than the throughput measured from the thread-based implementation for all atomicity and order semantics (See Figures 1, 2, and 3). The difference in the throughput between the two implementations increases with increase in update arrival rate until a saturation point is reached.

We measured average delivery times for four different update interarrival times. The average delivery time measured from the event-based implementation is smaller than

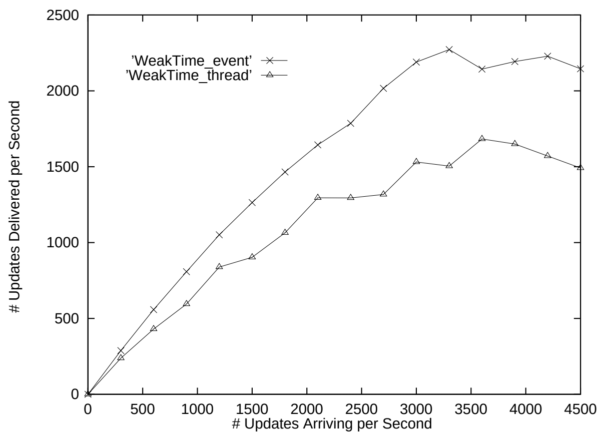

Figure 1. Throughput (Wk atm/Tm ord).

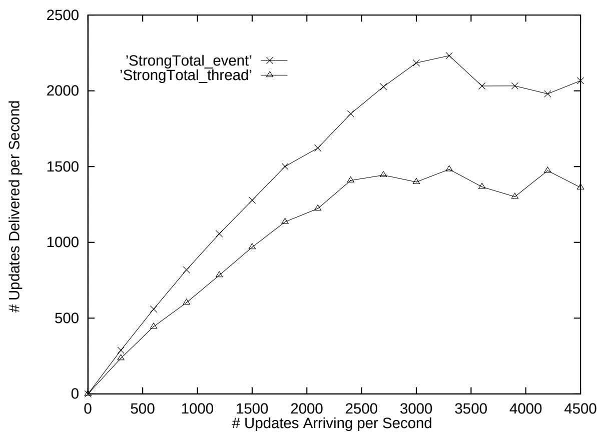

Figure 2. Throughput (Strn atm/Ttl ord).

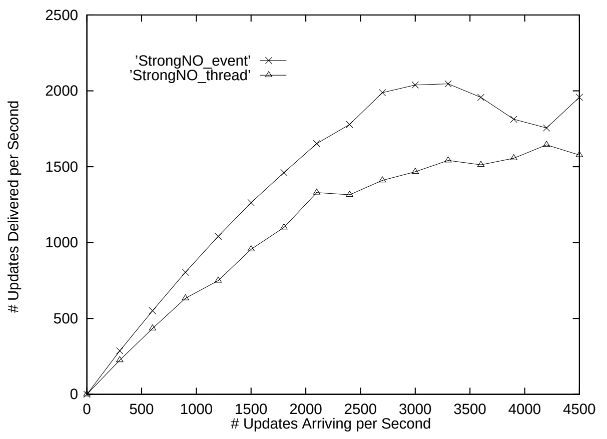

Figure 3. Throughput (Strc atm/No ord).

	Interarrival Time			
Event-based	2	5	8	10
Wk atm/Ttl ord	2.58	1.9	1.49	1.55
Strn atm/Tm ord	2.6	2.0	1.5	1.57
Strc atm/No ord	2.6	2.1	1.5	1.57
Strc atm/Ttl ord	2.62	2.13	1.52	1.6
Thread-based				
Wk atm/Ttl ord	2.87	4.0	4.0	4.2
Strn atm/Tm ord	2.9	4.1	4.1	4.21
Strc atm/No ord	2.88	4.0	4.1	4.2
Strc atm/Ttl ord	2.91	4.14	4.12	4.2

Table 1. Delivery time (msec)

the average delivery time measured from the thread-based implementation for all atomicity and order semantics for all four update interarrival times (See Table 1).

Similarly, the average stability time measured from the event-based implementation is smaller than the average stability time measured from the thread-based implementation for all atomicity and order semantics for all four update interarrival times(See Table 2).

Finally, the average number of messages exchanged per atomic broadcast is slightly higher in the event-based implementation than in the thread-based implementation (See Table 3).

The performance of the event-based implementation is almost always better than the thread-based implementation, except in case of number of messages exchanged. There are two sources of performance overhead in the thread-based implementation that are not present in the event-based implementation: context switching times between threads during scheduling and the execution time of synchronization primitives. On the other hand, there is one source of performance overhead in the event-based implementation that is not present in the thread-based implementation: the execution time of the `select()` system call.

An event occurrence results in a single execution of {`pthread_mutex_lock();pthread_mutex_unlock();`} and at least one thread context switch in the thread-based implementation. An event occurrence results in the execution of zero or one `select()` system call in the event-based implementation. The thread context switching time is about 16 microseconds (averaged over 10,000 `pthread_yield()` operations). The execution time of {`pthread_mutex_lock(); pthread_mutex_unlock();`} is about 14 microseconds (averaged over the execution of 10,000 such pairs). The execution time of the `select()` system call is extremely small (less than 5 micro seconds).

These measurements clearly show that the performance overhead due to the execution of the `select()` system call is far less than the performance overhead due to the execution of {`pthread_mutex_lock();`

Event-based	Interarrival Time			
	2	5	8	10
Wk atm/No ord	5.5	4.0	3.99	4.18
Wk atm/Ttl ord	5.5	4.9	4.2	4.22
Strn atm/Tm ord	5.51	4.9	4.22	4.22
Strc atm/No ord	5.5	4.9	4.2	4.21
Thread-based				
Wk atm/No ord	7.05	9.0	10.5	10.7
Wk atm/Ttl ord	8.0	9.5	9.6	10.6
Strn atm/Tm ord	8.2	9.6	9.6	10.8
Strc atm/No ord	8.1	9.7	9.6	10.7

Table 2. Stability time (msec)

	Interarrival Time			
	2	5	8	10
Event-based	2.0	2.9	4.0	5.0
Thread-based	1.9	2.5	3.0	3.3

Table 3. # of messages, (Wk atm/No ord)

`pthread_mutex_unlock();`} and one thread context switch. Perhaps, the difference in the performance between the two implementations can be reduced by a much more efficient implementation of a pthreads library. However, we believe that there are some fundamental overheads in the thread-based implementation that can't be avoided, and the performance of an event-based implementation of a group communication service is always expected to be better.

6. Conclusion

Both the thread-based and the event-based implementation techniques have been used extensively by researchers to implement group communication services. Our evaluation of these techniques suggests that an event-based implementation is preferable to a thread-based implementation. An event-based implementation results in better performance, simpler, more manageable, and portable code. It is worth noting that threads have been used in implementing a number of popular group communication services. There are two reasons that we can think of for the use of threads in implementing these services. First, a threads-based implementation is often considered a "natural" way of implementing systems that have concurrent events. This was the main reason why we used threads to implement group communication services in the past [7, 3]. Second, most of the group communication systems that use threads are designed to experiment with different features of a group communication service, such as flow control or establishing message stability. Perhaps, adding/removing features in an experimental system is simpler with threads than with events.

References

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM TOCS*, 13(4):311–342, 1995.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM TOCS*, 9(3):272–314, Aug 1991.

[3] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed Systems Engineering*, 4(2):109–128, Jun 1997.

[4] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *15th ACM PODC*, Philadelphia, PA, May 1996.

[5] S. Mishra, C. Fetzer, and F. Cristian. The timewheel asynchronous atomic broadcast protocol. In *1997 PDPTA*, pages 1239–1248, Las Vegas, NV, Jun 1997.

[6] S. Mishra, C. Fetzer, and F. Cristian. The timewheel group membership protocol. In *Proceedings of the Workshop on Fault Tolerant Parallel and Distributed Systems*, Orlando, FL, Apr 1998.

[7] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, Dec 1993.

[8] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in horus. In *14th ACM Symposium on Principles of Distributed Computing*, Aug 1995.