Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

S3F: THE SCALABLE SIMULATION FRAMEWORK REVISITED

David M. Nicol
Dong Jin
Yuhao Zheng

University of Illinois at Urbana-Champaign
Information Trust Institute
1308 West Main Street
Urbana, IL , 61801, USA

ABSTRACT

Following ten years of experience using the Scalable Simulation Framework (SSF), we revisited its API, making changes to better reflect use and support maintainability. This paper gives a quick overview of SSF, and discusses changes made in the second-generation API, S3F. Of particular interest is S3F's treatment of advancing time (in epochs), its treatment of "processes", and the support it gives the modeler over precise ordering of events that happen to be scheduled at the same simulation time.

1 INTRODUCTION

The Scalable Simulation Framework (SSF) was proposed over 10 years ago as an API for developing discrete-event simulations. A key motivation for SSF was to lay out an API such that models developed against it could be parallelized.

In its lifetime a number of implementations, variations, and extensions of SSF were developed, e.g., (Nicol, Liu, Liljenstam, and Yan 2003, Liljenstam, Nicol, Yuan, Yan, and Liu 2006, Liu, Mann, Van, and Hellman 2007). Numerous application models were used for research in networking, e.g., (Nicol, Goldsby, and Johnson 1999, Chen, Branch, Pflug, Zhu, and Szymanski 2005, Hao and Koppol 2003), in privacy and security, e.g., (O'Gorman and Blott 2007, Baek, Im, Park, Choi, and Jung 2004, Oh, Kim, Kim, Hong, and Sohn 2007, Yun, Sohn, Kang, and Lee 2007, Lee, Park, Jung, and Choi 2005, Zhao, Smith, and Nicol 2005, Liljenstam, Nicol, Berk, and Gray 2003), and others, e.g., (Walkowiak and Mazurkiewicz 2008). In that time we've learned a lot about using the SSF API, things that work, things that are rarely used, and learned a lot about maintaining and using a simulator in an academic context.

Revisiting the SSF API was long overdue. Motivated in part by the need for students and staff to be able to extend and modify a simulator for which there was expertise in-house, and motivated in part by the need for a network simulator that is particularly focused on supporting emulation, we rethought and redesigned the core SSF classes and interfaces; this paper sketches the new design.

The first problem arose, of course, what name? On reflection, any number of adjectives might be given to the the new system, adjectives beginning with the letter 'S'. Simple (perhaps). Second-generation. Standards-compliant. Safe, perhaps. And most probably, Slower. That would make SSSF, which gives one a headache to look at. We call it S3F.

2 A BRIEF OVERVIEW OF SSF

The Scalable Simulation Framework is an API developed to support modular construction of simulation models, in such a way that potential parallelism can be easily identified and exploited. It assumes the

implementing language will support objects of some kind, and has had realizations in Java, C++, Python, and Tcl.

The SSF API is minimalist and not tied to any particular domain where simulation models might be developed. Rather, it provides a base upon which domain-specific frameworks may be developed. For example, SSFNet is a framework that defines networking concepts (e.g., subnetworks, interfaces, hosts, protocols) and implements them using SSF objects. The SSFNet user interacts with the SSFNet interface, not so much with SSF.

SSF is built around five base classes. The `Entity` class is a container for simulation state and instances of other SSF classes. Classes `OutChannel` and `InChannel` model communication endpoints, carrying instances of the `Event` class to convey information. `OutChannel` methods are called to connect to particular instances of `InChannels` (multiple connections from one outchannel are permissible), and the `OutChannel::write` method is used to send an instance of the `Event` class to be received some time in the future, through instances of the `InChannel` class that are connected to the source outchannel. Instances of the `Process` class have code bodies that perform the simulation activity (and are "owned" by entities). Code execution is suspended when the process calls `Process::waitOn`, naming an inchannel to wait on, until an event is delivered to that inchannel.

The notion of a "timeline" is implicit in the SSF API, but is not a class to which the SSF user has any access. Every entity (and all the class instances it owns) are "aligned" to a timeline, which simply means that they all share the same event list, which serializes the execution of all simulation activity that occurs in processes owned by entities on the timeline.

Simulation time advances in SSF entirely due to temporal delays between when an event is written to an outchannel, and when that event is delivered to an inchannel. SSF provides a number of ways of specifying that delay, including a permanent lower bound that is declared when the connection between outchannel and inchannel is established. This lower bound is the key to finding parallelism in an SSF model, for consider—if $\delta > 0$ is the smallest lower bound on the latency of a channel with endpoints aligned to different timelines, then we are assured that any simulation that occurs on one timeline cannot affect the state of any entity on any other timeline within δ time; there is a temporal insensitivity between timelines that allows them to simulate in parallel within a window of time δ wide.

3 INTERFACE

The SSF view is that the user program creates the simulation model by calling constructors for various SSF objects, initializes them, then passes control to the simulation engine. Control is returned only after the simulation has entirely completed.

Early on we wanted to run an SSF model for a "time-step", e.g., a time during which wireless signal strengths could reasonably be assumed not to change much due to movement, and then deal with mobility and recalculation of received signal strengths. We made this work by putting this activity in between global barriers SSF was using for synchronization anyway, but it looked and felt like a hack. Later we could never get a GPU accelerated wireless channel computation integrated with PRIME because structure used by the GPU framework had to be embedded within PRIME, but needed to be accessed outside of PRIME.

S3F takes a slightly different approach. From the thread of control that starts at `main` the code eventually creates a S3F object called an `Interface`. The base class constructor for `Interface` creates a number of pthreads for the simulation; S3F usage requires definition of a derived class to instantiate some virtual functions and possibly override some default functions. Thereafter, the control thread (now also a pthread) and the simulation threads coordinate through barrier synchronizations. In particular, after the simulation model has been built and initialized, the control thread will tell the simulation threads to advance simulation time up to some time it specifies, or until some global condition of state is met. It frees them to advance the simulation that far, and waits until they have completed. Once they have, the simulation state lays dormant while the control thread may do any one of a number of things, e.g., acquire captured traffic from virtual machines running applications, traffic whose routing is to be simulated by the

simulator, and/or interact with other pthreads it has spawned. When the control thread has accomplished what it needs it may direct the simulation to advance again. Figure 1 illustrates the architecture of this approach.

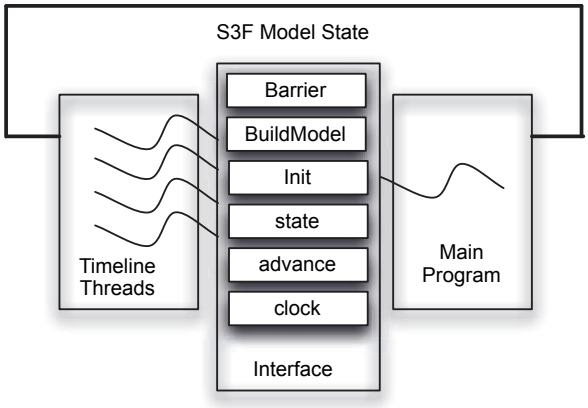

Figure 1: Logical Organization of S3F Simulation Program.

A thumbnail sketch of the approach is as follows.

1. The control thread creates an instance of a class derived from `Interface`.
2. The derived class constructor passes its parameters to the base `Interface` class constructor. This creates a number of pthreads whose code bodies are the `Timeline::thread_function`. These all immediately synchronize on a barrier that includes them all, and the control function.
3. The control thread continues, and calls `Interface::BuildModel`. This actually runs code written by a modeler or simulation framework builder, and does whatever it likes to create the structure of a simulation model by calling constructors of various S3F objects.
4. After `Interface::BuildModel` returns, the control thread calls `Interface::Init`. This method just calls the `init` method on every S3F "Entity" class instance, methods written by the modeler.
5. The control now calls `Interface::advance` with arguments specifying how/when the newly initiated simulation epoch should end. The control thread suspends, the Timeline threads are released to advance simulation time until the stopping criterion is met, and when they are done the control thread is released while the Timeline threads are suspended.

One condition for ending an epoch is that each timeline has advanced to a simulation time some common Δ time in the future. S3F also supports general specification of a condition which causes a timeline to stop. That condition is expressed by an `Interface` method that returns a boolean value indicating whether the state of entities aligned to the timeline satisfy a stopping condition. This method is called at the end of every synchronization window (described in more detail later), with the value of a minimum and maximum reduction returned. A decision to continue on may be made based on whether *any* timeline has encountered a stopping condition, or *all* timelines have. To avoid the negative consequences of some timeline *never* returning true, this technique is always used in conjunction with a maximal epoch length Δ beyond which no timeline will advance. So long as the stopping condition is monotone (meaning that once it becomes true it remains true, until some state change caused by the control thread outside of the normal flow of simulation events), we can guarantee that the simulation will be stopped immediately after the first synchronization window in which the stopping criterion is globally satisfied.

4 S3F OBJECTS

The S3F modeler has access to a relatively small number of base classes for S3F objects. They are for the most part similar to those of SSF. There are some differences though, which we will highlight.

4.1 Time Types

SSF gave time the type `ltime_t`, and S3F follows suit. SSF allowed compile-time selection of the native C++ or user-defined type. By contrast S3F insists that `ltime_t` be some sort of integer, preferably a long integer. The time-scale of the clock counter is specified at the time the `Interface` instance is created. The reasoning behind the decision is an emphasis in applications driving S3F development to have precise modeler control over the ordering of submodel executions. Roundoff strategies in floating point computation make such control very difficult to guarantee.

Double precision numbers are particularly useful for generating continuous random variables, as well as other calculations that ultimately have to be interpreted in units of simulation time. S3F provides routines to convert `ltime_t` to double, and back. The units of the doubles involved may be identical to those of the simulation clock, or may be specified separately and be rescaled as part of the conversion.

4.2 Entities and Processes

One ought to think of and SSF or S3F simulation as interactions among a number of `Entity` objects. The simulation state is distributed among entities, entity methods provide the logic of the simulation as they modify the simulation state, the entities communicate with each other. The high level S3F view of `Entity` is very much like the SSF view. Each entity is *aligned* to some `Timeline` object (discussed below), and is "owner" of instances of other simulation objects, `OutChannel`, `InChannel`, and `Process`.

Alignment of `Entities` is potentially dynamic in SSF. The things on which they were aligned—timelines—are not accessible to the user. When an SSF `Entity` is created it is by default aligned to its very own timeline, but you can then re-align it onto the timeline of another `Entity`. Dynamic realignment in the course of the simulation was permitted by the API, but was rarely supported and even more rarely used.

Correspondingly S3F takes a more static view of alignment. The `Timeline` class is visible to the modeler (although creation of a `Timeline` is reserved for the default `Interface` class constructor). An S3F `Entity`'s constructor is told on which `Timeline` it is aligned. There is no mechanism built in for dynamically re-aligning entities, as yet.

Under SSF the only code body that acts on simulation state is the `action` method of the `Process` class. Under S3F any `Entity` method that returns `void *` and takes an `Activation` argument (described later) may serve. Under S3F the `Process` class still exists, but most of its functionality has been moved to the `Entity` class.

SSF views a `Process` as a logical thread expressed in method `Process::action`. To remain ever-present, the main body of `action` must be written inside of an eternal "while(1) ..." loop. When `action` executes it can change state data, create and send "Events", and ultimately suspend until re-activated. The two basic means of suspension are `waitFor` and `waitOn`. The former takes an `ltime_t t` argument and suspends execution of the code body at that point until the simulation clock advances exactly `t` units of simulation, at which point execution resumes—again, at that point in the code. A `waitOn` argument names an `InChannel`; `action` is suspended at the `waitOn` call until such simulation time as an "Event" is delivered to the `InChannel`. Execution resumes at that point, and continues until `action` executes another call that suspends it.

One of the biggest differences between SSF and S3F is in the semantics of executing Processes. In our C++ implementations of SSF we created our own lightweight threading, via source-to-source translation and dependence on the simulation modeler using the right pragmas in all the right places. Building our own threads was tricky, and use of a source-code level debugger trickier still. We found that tricks we used to

hide the translation during debugging were fragile, often breaking when g++ was updated. With experience we came to the conclusion that the benefits of supporting arbitrary suspension within a `Process` was not worth the support effort.

Another problem with respect to code maintenance was the C++ implementation's reliance on third party packages, e.g., for fast memory allocation. So while there are various academic projects developing threads for C++, we decided that S3F would use something supported natively in the GNU compiler g++. That leaves us with pthreads, which are a bit heavy-weight for code snippets in simulations.

One of the work-arounds to high thread overheads that SSF allowed for was "simple processes". A process was simple, basically, if any call that would suspend the process was placed in the code so that when the suspended thread was re-animated the code body would go immediately to the top of the encompassing "while(1)" loop. In a simple process the states of local variables did not have to be preserved across suspending calls, and so their implementations did not have to use threading of any kind. Still, it was sometimes a challenge to express the logic of a process to make it a simple one.

Because of these factors S3F takes an approach that preserves normal C++ run-time semantics, a decision that makes it straightforward to integrate C++ libraries of supporting code. S3F has a `Process` class, but this serves mostly as a container for a pointer to a `Entity` that owns it, and a method of that entity which serves as its code body. Every execution of the `Process` body is effectively a subroutine call to that method, made from within the simulation scheduler. Therefore, every execution of the `Process` body begins at the first line of that method. To be sure, one can save state in the Entity that owns the `Process` or the `Process` itself (if the modeler creates class derived from the `Process` base class) that directs the control flow from that point differently depending on the input presented to the `Process` and the saved state. The main point is that ordinary C++ run-time semantics apply whenever a `Process` is called to execute.

Like SSF, S3F provides `waitFor` and `waitOn` calls; while both are `Process` methods in SSF, in S3F `waitFor` is an `Entity` method and `waitOn` is an `InChannel` method. The new `waitFor` is different in a variety of ways.

- Optionally, a `waitFor` call may specify *a different process* to execute after the elapsed simulation time. This is possible because we have decoupled the notion of suspension from the `waitFor` call altogether—in SSF it means to suspend *the executing process*. In S3F an executing process can schedule the execution of another as a timer, for example. There is a restriction though that the entity that owns the process scheduled by `waitFor` has to be aligned to the same timeline as the owner of the process making the call.
- Optionally, the call may provide an `Activation` as input to the scheduled process call. We will say more about `Activation`s later. For now it is enough to know that it is a smart pointer wrapper around a message.
- Optionally, the call may provide an unsigned integer *priority* to the call. All executions of `Process` bodies are ordered by simulation time, but priorities may be included to order the execution of processes scheduled to execute at precisely the same time. As with simulation time, the smaller the value, the higher the priority. S3F allows 15 bits of priority to be expressed by the user (and reserves some bits for itself).
- On creation, a process can be given a priority (or is given a default priority); methods exist to read and change it dynamically. The process's own priority is used at the instance a `waitFor` is called, if the `waitFor` call does not itself specify a priority.

Experience with SSF highlighted the utility of allowing a user to create a global name space where entities and inchannels may be identified with text strings. The original motivation was construction of models across machines that do not share memory. When an S3F Entity is constructed one may optionally provide a string name. The name needs to be unique among all Entities created in the model. The string

is entered into a static `Entity` class directory, with a method provided for looking up the address of an entity, given its string name.

4.3 Timeline

A `Timeline` hosts an event list used for scheduling, and is responsible for advancing the simulation related to all entities aligned to itself. All of the underlying logic of event execution is contained in the `Timeline`, but is hidden from the user. The pubic interface to the `Timeline` class is very limited. The user can obtain its current simulation time (basically the time stamp of the current event being processed, or a time before which no future event will ever execute), and the time-scale of the clock ticks. It also reports its `event horizon`, which is a technical notion we'll save for the discussion on synchronization.

4.4 InChannel

The S3F notion of an `InChannel` is very much like SSF's, although the mechanics of using them is a little different. In both systems it serves as an endpoint for a communication channel. In both systems one can use `waitOn` to tell the simulation to awaken some process when a message is delivered to the `InChannel`. S3F continues the SSF notion that delivery is taken of a message that arrives to an inchannel at simulation time t only by those processes that have attached themselves to that inchannel. There is no buffering. A message may arrive to an inchannel with no processes listening, and if it does it is silently discarded.

S3F changes the way processes are attached to inchannels. Under SSF `waitOn` was a `Process` method, and the call identified one or more inchannels the process would then suspend on. Delivery of a message to any one of those inchannels would unsuspend the process.

Under S3F `waitOn` is an `InChannel` method. It has one argument (which may be implicit), the process that should be activated when a delivery is made to the inchannel. The entity that owns the process so identified must be on the same timeline as the owner of the InChannel. Like SSF, in S3F multiple processes may simultaneously be waiting on the same inchannel, and like SSF, in S3F all of those processes are activated as a result of a delivery, each one being given access to the accompanying message.

In the semantics of SSF, the attachment of a process to an inchannel disappeared just as soon as the `waitOn` call returns. S3F allows this interpretation, but also allows the binding to persist past the execution of the process body resulting from an attachment. If a process is attached to an inchannel by a `waitOn` call and as a result of an arrival the process is executed, the process will **not** be called on the next arrival **unless** it is passed to the inchannel again by another `waitOn` call. However, one can make the attachment persist across process executions by calling `InChannel` method `bind`. Like `waitFor` the argument to `bind` is a process (either explicit, or implicitly the process executing when the call is made). Like `waitOn`, the owner of the process being bound must be on the same timeline as the owner of the inchannel.

In addition to `waitFor` and `bind`, S3F provides methods for canceling prior calls to these methods, and for querying whether a given process is either bound or waiting on the channel. Like entities, inchannels may have string names declared for them at time of construction. Unlike entities though, the inchannel name has to be unique only among inchannels on the same entity. This makes it easy, for example, to associate an inchannel with some well known interface, e.g., port80. Coupled with textual entity names, we have a global name space for identifying an inchannel—the entity name paired with the inchannel name. The utility of such a name space is better appreciated in the context of the `OutChannel::mapto` function, discussed below.

4.5 Message and Activation

SSF uses the `Event` base class as the basis for communication. Methods involved with the sending and receiving pass pointers to the events. This design decision was fine for Java where garbage collection is done automatically, but was a nightmare for the C++ implementation. A sent message may be silently

duplicated for multiple destinations. It is expensive to have the system make unique copies of each message, but if the message is shared, it is complicated to delete it once it is needed nowhere in the system. Our C++ implementations created their own schemes to do reference counting, but these still required active modeler specification of when a reference to an event was no longer needed.

Now ten years later a great deal of thought and effort has gone on in the C++ community about smart pointers. The current standard—embedded already in the GNU g++ compiler—is the `shared_ptr` smart pointer in the `std::tr1` namespace.

Oddly, an "Event" in SSF is actually a message, and after years of explaining this we changed the class name to `Message`. The base class constructor allows one to pass along (and carry along) an integer type for the message, the utility of this will be noted shortly.

S3F contains a typedef using the term `Activation` to refer to a `shared_ptr<Message>`. For a communication to have any significant content at all, the user needs to create a class derived from `Message` to carry what he wants. A user that wants to send a message of type `MyMessage` (derived from `Message`) creates an instance of it, e.g., suppose `mmsg` is a pointer to a `MyMessage`. He creates an Activation such as `Activation act(mmsg)`, and passes `act` *by value* in the communication. Likewise on receipt an activation is passed by value. The passing by value is the magic part needed for reference counting.

When a process is activated owing to a prior `waitOn` (or even a `waitFor`) its code body is passed one argument, an activation. C++ does not have introspection, all the code body knows is that the activation wraps a pointer to a `Message`. Downcasting is possible, but downcast to what subclass? If the message is from a derived class of `Message` there is no native way of inferring that. S3F helps in part by having `Message` carry a "type" code, offered at creation. If the recipient of an activation does not know what the derived type is by context, it can get the base class type, so that the modeler can create a code mapping integers to subclasses.

The beauty of the thing is that activation wrapped messages can be placed in STL containers, passed from one routine to another, have multiple wrappers of a common message be delivered to multiple inchannels—even multiple timelines—and the protected message will be automatically deleted when—and only when—the last activation protecting it has past out of scope. User involvement in reference counting is minimized.

4.6 OutChannel

An instance of the `OutChannel` class is the "send" point for an Activation. The constructor has a mandatory argument—a pointer to the Entity that "owns" it, and an optional argument to be discussed later. The main two methods are `write`, and `mapto`. Both have many variations that work through all combinations of optional arguments.

`write` initiates the transfer of an activation to one or more inchannels (which we will see are specified by `mapto` calls). The one required argument is an instance of an activation. Optionally one may include a "per-write" time increment. The optional second argument of the `OutChannel` constructor is a lower bound on what value may be given as a "per-write" increment. We will comment more deeply on the time increment later. Here we just accept that if the per-write increment is not given, it automatically takes value 0.

A `mapto` call has two parameters, a pointer to an inchannel, and a time delay. It is a declaration that that every write made to the outchannel will be delivered to the the named inchannel. The time delay is known as the *transfer time*, and is involved in computing the time at which the activation presented by a write will be delivered to the inchannel.

If an inchannel I is targeted by a `mapto` with transfer delay d and at time t a call is made to the outchannel's `write` method with per-write delay w, then the arrival time of the activation at an inchannel is $t+d+w$. The motivation for this approach lies in S3F's support for parallelism. w is a lower bound on the delay between the sending of any activation from the outchannel to the inchannel. Indeed, if the outchannel

was constructed with an optional lower bound b on the per-write delay, then $b+w$ is a lower bound. The synchronization mechanism uses this kind of information to establish synchronization windows.

While the factoring of the delay into two parts may seem odd, it has its uses beyond support of parallel simulation. If the outchannel to inchannel mapping represents a communication line between two switches, one component of the the latency is due to that line's bandwidth, a component that is fixed. A second component is due to queueing delay. When the queue is FCFS, the arrival time of the activation can be computed at the time the message joins the queue, e.g., is "written". Here the per-write delay would be computed as the time in queue before the message is sent.

5 PRIORITIES

One of our objectives in S3F was to ensure that the modeler has the tools necessary to exercise complete control over the ordering of process body executions. Of course these are ordered by simulation time, the issue is control over ordering of executions scheduled at exactly the same simulation time. This control is especially important to time-stepped models where time advances globally in discrete steps.

We order execution of process bodies scheduled at the same time by using a second key in the event list structure. If two events have the same first key—simulation time—the one with a smaller second key is executed first (the same ordering as with the first key). We give the modeler control over allocation of the key, the event priority. If a modeler is careful to ensure that no two process body executions scheduled to occur at the same time have the same priority, then there are no uncertainties. If two process executions happen to be scheduled at the same time, but with the same priority, an "invisible" final priority is applied. Each timeline increments a counter when it creates an event to schedule, and the value of that counter at time of the event's creation adjudicates. As with the first two keys, a smaller counter value takes precedence over a larger one. A careful modeler can use knowledge of the third key priority, but it should be noted that an event's third key comes from the timeline one which it was created, and that events on a timeline's event list may have their origins in different timelines.

Process body executions occur either as the result of a `waitFor` statement, or as the result of delivering an activation to an inchannel. In the former case, a modeler may specify the execution's priority as part of the `waitFor` call; if no priority is explicitly given, the scheduler assumes a priority value associated with the process itself.

To understand how priorities play a role in the execution of processes that react to an activation arriving at an inchannel, it is helpful to go through what actually happens. When a user specifies a priority p at any of the interfaces the API permits, the system immediately transforms it. $(p+1)$ is shifted left by 8 bits, and then is AND-masked with `0xffffff`. This creates a range $0-255$ of values lower than any transformed user-supplied priority, which allows the system to schedule events ensured to happen first, among all events with the same time-stamp.

Processing of the delivery of an activation to an inchannel is one of these "happens first" events. The system takes all processes either bound to the inchannel or waiting on the inchannel, and for each schedules an execution at this same time. If a user priority was passed when the process was bound to the inchannel (or declared that it is waiting on the inchannel), then the transformed priority is embedded in the event. If no such priority was declared when the process became attached to the inchannel, then the process's own priority at this instant, transformed, is used as the event's priority.

The net effect of this design is that process executions caused by `waitFor` calls and ones caused by receipt of an activation on an inchannel are handled in the same way: the modeler has control over the priority used to order events scheduled at the same time, and the priority is specified by a process aligned to the same timeline where the eventual processing will occur. The simplicity of the priority model makes it straightforward for a modeler to design a priority scheme that accomplishes his objectives.

6 SYNCHRONIZATION

S3F supports parallel execution, which requires synchronization among the timelines. Much of the motivation and design of S3F is to support synchronization more-or-less transparently, yet provide hooks to the sophisticated modeler to transfer modeling information to the synchronization engine that is used to improve performance.

The basic idea behind synchronization is simple. Use information about latencies across communication paths established between outchannels and inchannels to establish a window of simulation time with the property that no activation written to an outchannel at a time within that window will be received by an inchannel *on a different timeline* at a time also within that window. The windows are implemented with barrier synchronizations. The upshot is that writes whose activations cross between timelines do not have to be immediately delivered—their receipt lies on the other side of a barrier synchronization, so they can be buffered pending a step where synchronized timelines exchange such activations, and integrate them into their target timeline event lists.

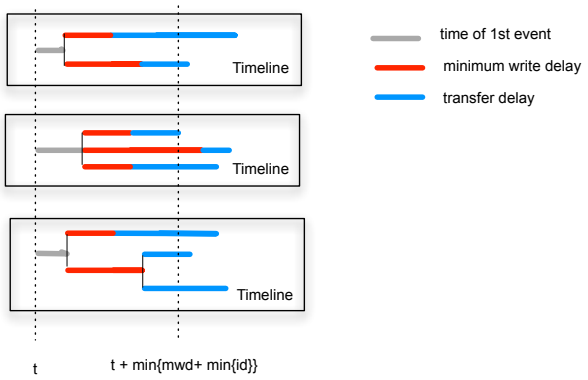

Figure 2: Synchronization Window.

Figure 2 illustrates the concept. Suppose the timelines have all advanced to simulation t, and have in their event lists all events known (at that time) to be executed. Execution of some of these events may of course introduce other events into the list, but every future event known at time t is in the event list of the timeline that will execute it. The timelines are working to coordinate how far ahead in simulation time they can safely advance. Each timeline identifies the time of the first event it has in its list, which is a lower bound on when next the timeline will execute a process that performs a write that crosses timeline boundaries. Each timeline also identifies the minimum over all mapped cross-timeline outchannel/inchannel pairs of the outchannel's minimum per-write delay, and the transfer delay to the inchannel. That is, each timeline i identifies a lower bound on the arrival time of any future cross-timeline write as

$$L_i = n_i + B_i$$

where we separate the bound into time of next event n_i, and

$$B_i = \min_{\textit{outchannel } c} \left\{ w_c + \min_{\textit{X timeline mapped inchannels } k} t_{c,k} \right\}$$

where w_c is the minimum per-write delay declared for outchannel c, and $t_{c,k}$ is the transfer time between c and inchannel k owned by an Entity aligned to a different timeline than c's owner. To establish the synchronization window the timelines offer their respective L_i values to a global min-reduction. On being released from the associated barrier, each timeline can read the minimum among all offered values—this is the upper edge of the window. Simulation time may advance to one clock tick less than this value.

The value of n_i may change from window to window, but B_i need not, at least so long as there are no changes in the inchannels mapped, the transfer delays, or the minimum per-write delays, the result of a prior computation can be used. However one cannot always expect the set of mapped outchannels/inchannels to remain constant, and changes in model state may allow (or require) the model to change an outchannel's per-write minimum delay, or some transfer delay. As we have seen already, S3F allows dynamic `unmap` and `mapto` calls, and dynamic changes to minimum per-write and transfer delays. S3F tries to minimize the impact of those changes by being smart about recomputing B_i. Once computed, S3F also counts the number of cross timeline outchannel/input pairs that actually achieve the computed value of B_i. Any change in mapped channels or their delays that decrease that count need not trigger a recomputation so long as the change leaves at least one outchannel/input pair with B_i's value. Likewise any change that cannot possibly lower B_i (e.g., *increasing* either the per-write minimum delay or a transfer delay) will not trigger recomputation of B_i. However, requests for changes that do affect B_i raise a flag that later triggers re-computation of B_i.

Calls to outchannel methods `mapto`, `new_transfer_delay` and `new_min_write_delay` may request changes that cannot be safely implemented immediately. These requests are queued to be processed in FCFS order after a timeline has executed all events in the current event window. FCFS processing implies that within a window a number of changes to a channel might be requested and buffered, but for both minimum per-write delay and transfer delay, only the last change requested "takes".

Processing of queued delay change requests may raise the flag that B_i must be recomputed. After the change requests are processed, that flag is tested, and if true, S3F does a complete analysis of the timeline's delays, and recomputes B_i.

After a timeline has processed all the window's events, implemented any buffered delay changes, and recomputed B_i (if needed), it sets its clock to the time of the window edge minus one clock tick, and enters another barrier synchronization to wait for all other timelines to do so also. When they are released, they transfer buffered events that resulted from cross timeline writes into their event lists (thereby ensuring that each time of next event n_i is what it needs to be), and compute the upper edge of the next synchronization window.

7 PERFORMANCE

We are porting the SSFNet concepts to S3F and performing preliminary tests of performance compared with SSF. On a model with 1500 hosts, running 500 intensive UDP flows (models of streaming video), for 10 simulation seconds on a multi-cpu host we observe that with small numbers of threads (1-4) the average performance of the two are comparable, with the distinction that the SSF based runs exhibit considerably more variance than the S3F runs. Both systems deliver close to 0.5M events per second. As the number of cpus used increases though, the S3F performance flattens out, due to a an-easy-implement but non-scalable barrier synchronization algorithm. We are actively engaged in performance tuning and are confident that since the core mechanisms are shown to be equally as fast as SSF (on one cpu), we can optimize the synchronization and communication overheads.

8 CONCLUSION

We developed the second generation API of the Scalable Simulation Framework, dubbed S3F, to address issues of maintainability and use, based on more than ten years of experience with SSF. The main distinguishing differences are

- S3F uses only standard language and library features found in the GNU C++ compiler, g++. In particular, S3F eschews the user-level threads found in the C++ implementations of SSF, eschews experimental third party libraries for memory management, and uses the Standard Template Library (STL) extensively. These decisions significantly enhance our ability to maintain S3F in an academic environment.

- S3F's architecture allows time advancement to be broken up into epochs. At the end of one such, control is released from the simulation threads to allow other activity (which may be based on and alter the simulation state), for activity such as recalculation of received radio strength maps and location of mobile devices. The epoch may be a simple interval of time whose length is declared at its beginning, or one whose end is discovered, as a function of evolving simulation state.
- S3F gives a modeler extreme control over the ordering of process body executions. Simulation time advance is required to be in discrete clock ticks (whose units of scale are declared at initialization), and the API supports comprehensive control over priorities used to order executions scheduled at the same instant in simulation time.

It is relatively straightforward to port existing SSF-based framework code (such as SSFNet) to run on top of S3F, and early experience doing that suggests that S3F can be tuned to have comparable performance as SSF. Future work includes extending the API to encompass distributed memory, and integrating a number of emulation systems (e.g., OpenVZ) into an S3F-based framework.

ACKNOWLEDGEMENTS

This material is based upon worked supported under Dept. of Energy under Award Number DE-0E00000097, under support from the Boeing Corporation, and under support from Rockwell-Collins Corporation. This report was prepared as an account of work sponsored in part by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

Baek, C., E. Im, E. Park, K. C. Choi, and G. Jung. 2004. In *Proceedings of the 6th International Conference on Advanced Communication Technology*, 312–316.

Chen, G., J. Branch, M. Pflug, L. Zhu, and B. Szymanski. 2005. "Sense: A Wireless Sensor Network Simulator". In *Advances in Pervasive Computing and Networking*, edited by B. K. Szymanski and B. Yener, 249–267. Springer US.

Hao, F., and P. Koppol. 2003. "An Internet scale simulation setup for BGP". *ACM SIGCOMM Computer Communication Review* 33 (3): 43–57.

J. Liu, S. M, N. Van, and K. Hellman. 2007. "An Open and Scalable Emulation Infrastructure for Large-Scale Real-Time Network Simulations". In *Proceedings of the 26th IEEE International Conference on Computer Communications*, 2476–2480.

Lee, J., S. Park, G. Jung, and K. Choi. 2005. "Simulation framework for cyber terrors and defense". In *Proceedings of the 9th WSEAS International Conference on Communications*, 79:1–79:5. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS).

Liljenstam, M., D. Nicol, Y. Yuan, G. Yan, and J. Liu. 2006, Jan.. "RINSE: the Real-time Interactive Network Simulation Environment for Network Security Exercises". *Simulation : Transactions of the Society for Modeling and Simulation International* 82 (1): 43–59.

Liljenstam, M., D. M. Nicol, V. Berk, and R. Gray. 2003, October. "Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing". In *Proceedings of the 2003 Workshop on Rapid Malcode (WORM)*, 24–33. Washington, DC.

Nicol, D., M. Goldsby, and M. Johnson. 1999. "Fluid-based simulation of communication networks using SSF". In *Proceedings of the 1999 European Simulation Symposium*, Volume 2. Citeseer.

Nicol, D. M., J. Liu, M. Liljenstam, and G. Yan. 2003. "Simulation of large scale networks I: simulation of large-scale networks using SSF". In *Proceedings of the 2003 Winter simulation conference*, 650–657, edited by S. Chick and P. J. S
'anchez and D. Ferrin and D. J. Morrice: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

O'Gorman, G., and S. Blott. 2007. "Large scale simulation of Tor: modelling a global passive adversary". In *Proceedings of the 12th Asian computing science conference on Advances in computer science: computer and network security*, ASIAN'07, 48–54. Berlin, Heidelberg: Springer-Verlag.

Oh, Y.-K., W. N. Kim, D. S. Kim, S. J. Hong, and K. W. Sohn. 2007. "Design of Actor based Worm Modeling System Using DML.". In *IPC'07*, 210–213.

Walkowiak, T., and J. Mazurkiewicz. 2008. "Functional Availability Analysis of Discrete Transport System Realized by SSF Simulator". In *Computational Science ICCS 2008*, edited by M. Bubak, G. van Albada, J. Dongarra, and P. Sloot, Volume 5101 of *Lecture Notes in Computer Science*, 671–678. Springer Berlin / Heidelberg.

Yun, J., K. Sohn, H. Y. J. Kang, and D. Lee. 2007. "Dynamic Simulation on Network Security Simulator Using SSFNET". *Convergence Information Technology, International Conference on* 0:1168–1172.

Zhao, M., S. Smith, and D. Nicol. 2005, November. "Aggregated Path Authentication for Efficient BGP Security". In *Proceedings of the 2005 ACM Conference on Computer and Communications Security (CCS 2005)*, 128–138. Alexandria, VA.

AUTHOR BIOGRAPHIES

DAVID M. NICOL is Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, and is Director of the Information Trust Institute. He holds a B.A. in mathematics from Carleton College (1979), and M.S. and Ph.D. degrees in computer science from the University of Virginia (1983,1985). Prior to joining UIUC, he taught at the College of William & Mary, and Dartmouth College. He has served in many roles in the simulation community (e.g., Editor-in-Chief of ACM TOMACS, General Chair of the Winter Simulation Conference Executive Board of the WSC), was elected Fellow of the IEEE and Fellow of the ACM for his work in discrete-event simulation, and was the inaugural recipient of the ACM SIGSIM Distinguished Contributions award. His current research interests include application of simulation methodologies to the study of security in computer and communication systems. His email address is dmnicol@illinois.edu.

DONG JIN is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He holds a B.Eng. with first class honors in computer engineering from Nanyang Technological University, Singapore (2005), and a M.S. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign (2010). His research interests lie in the areas of computer security, large-scale computer and communication system modeling and simulation. His email address is dongjin2@illinois.edu.

YUHAO ZHENG is a Ph.D. student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests lie in the areas of computer security, large-scale computer and communication system modeling and simulation. His email address is zheng7@illinois.edu.