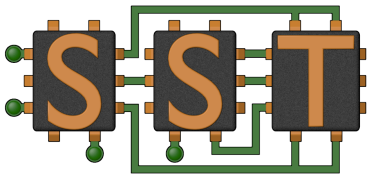
SST/macro 8.0: Developer's Reference

Sandia National Labs
Livermore, CA

April 17, 2018



Contents

1 Introduction **2**

1.1 Overview . . . 2

1.2 What To Expect In The Developer's Manual . . . 2

1.3 Use of C++ . . . 3

1.4 Polymorphism and Modularity . . . 3

1.5 Most Important Style and Coding Rules . . . 4

2 SST/macro Classes **5**

2.1 Factory Types . . . 5

2.1.1 Usage . . . 5

2.1.2 Base Class . . . 6

2.1.3 Child Class . . . 7

3 SST/macro Connectable Interface **9**

3.1 Required Functions . . . 9

3.2 Example External Component . . . 10

3.2.1 Python configuration . . . 11

3.2.2 Makefile . . . 11

4 SProCKit **12**

4.1 Debug . . . 12

4.2 Serialization . . . 13

4.3 Keyword Registration . . . 15

5 Discrete Event Simulation **16**
5.1 Event Managers . . . 16
5.1.1 Event Handlers . . . 17
5.1.2 Event Heap/Map . . . 18
5.2 Event Schedulers . . . 18

6 Software Models **19**
6.1 Applications and User-space Threads . . . 20
6.1.1 Thread-specific storage . . . 22
6.2 Libraries . . . 22
6.2.1 API . . . 23
6.2.2 Service . . . 24
6.3 Distributed Service . . . 24
6.3.1 Coordinating servers and clients . . . 25

7 Hardware Models **27**
7.1 Overview . . . 27
7.2 Connectables . . . 30
7.3 Interconnect . . . 31
7.4 Node . . . 32
7.5 Network Interface (NIC) . . . 32
7.6 Memory Model . . . 33
7.7 Network Switch . . . 33
7.8 Topology . . . 34
7.8.1 Basic Topology . . . 34
7.9 Router . . . 35

8 A Custom Object: Beginning To End **37**

9 How SST/macro Launches **42**
9.1 Configuration of Simulation . . . 42
9.2 Building and configuration of simulator components . . . 43
9.2.1 Event Manager . . . 43
9.2.2 Interconnect . . . 44
9.2.3 Applications . . . 44
9.3 Running . . . 45

10 Statistics Collection **46**

10.1 Setting Up Objects 46

10.2 Dumping Data 47

10.3 Reduction and Aggregation 47

10.4 Storage Contraints 48

Chapter 1

Introduction

1.1 Overview

The SST/macro software package provides a simulator for large-scale parallel computer architectures. SST/macro is a component within the Structural Simulation Toolkit (SST). SST itself provides the abstract discrete event interface. SST/macro implements specifically a coarse-grained simulator for distributed-memory applications.

The simulator is driven from either a trace file or skeleton application. Simulation can be broadly categorized as either off-line or on-line. Off-line simulators typically first run a full parallel application on a real machine, recording certain communication and computation events to a simulation trace. This event trace can then be replayed post-mortem in the simulator.

For large, system-level experiments with thousands of network endpoints, high-accuracy cycle-accurate simulation is not possible, or at least not convenient. Simulation requires coarse-grained approximations to be practical. SST/macro is therefore designed for specific cost/accuracy tradeoffs.

The developer's manual broadly covers the two main aspects of creating new components.

- Setting up components to match the `connectable` interface linking components together via ports and event handlers
- Registering components with the factory system to make them usable in simulation input files

1.2 What To Expect In The Developer's Manual

The developer's manual is mainly designed for those who wish to extend the simulator or understand its internals. This user's manual, in contrast, is mainly designed for those who wish to perform experiments with *new* applications using *existing* hardware models. The user's manual therefore covers building and running the core set of SST/macro features. The developer's manual covers what you need to know to add new features. The SST design is such that external components are built into shared object `.so` files.

1.3 Use of C++

SST/macro (Structural Simulation Toolkit for Macroscale) is a discrete event simulator designed for macroscale (system-level) experiments in HPC. SST/macro is an object-oriented C++ code that makes heavy use of dynamic types and polymorphism. While a great deal of template machinery exists under the hood, nearly all users and even most developers will never actually need to interact with any C++ templates. Most template wizardry is hidden in easy-to-use macros. While C++ allows a great deal of flexibility in syntax and code structure, we are striving towards a unified coding style.

Boost is no longer required or even used. Some C++11 features like `unordered_map` and `shared_ptr` are used heavily throughout the code.

1.4 Polymorphism and Modularity

The simulation progresses with different modules (classes) exchanging events or messages. In general, when module 1 sends a message to module 2, module 1 only sees an abstract interface for module 2. The polymorphic type of module 2 can vary freely to employ different physics or congestions models without affecting the implementation of module 1. Polymorphism, while greatly simplifying modularity and interchangeability, does have some consequences. The "workhorse" of SST/macro is the base `event` and `message` classes. To increase polymorphism and flexibility, every SST/macro module that receives events does so via the generic function

```
1 void handle(event* ev){
2 ...
3 }
```

The prototype therefore accepts *any* event type. The class `message` is a special type of event that refers specifically to a message (e.g. MPI message, flow in the context of TCP/IP) carrying a complete block of data or file. Misusing types in SST/macro is *not* a compile-time error. The onus of correct event types falls on runtime assertions. All event types may not be valid for a given module. A module for the memory subsystem should throw an error if the developer accidentally passes it an event intended for the OS or the NIC. Efforts are being made to convert runtime errors into compile-time errors. In many cases, though, this cannot be avoided. The other consequence is that a lot of dynamic casts appear in the code. An abstract `event` type is received, but must be converted to the specific message type desired. NOTE: While *some* dynamic casts are sometimes very expensive in C++ (and are implementation-dependent), most SST/macro dynamic casts are simple equality tests involving virtual table pointers.

While, SST/macro strives to be as modular as possible, allowing arbitrary memory, NIC, interconnect components, in many cases certain physical models are simply not compatible. For example, using a fluid flow model for memory reads cannot be easily combined with a packet-based model for the network. Again, pairing incompatible modules is not a compile-time error. Only when the types are fully defined at runtime can an incompatibility error be detected. Again, efforts are being made to convert as many type-usage problems into compiler errors. We prefer simulation flexibility to compiler strictness, though.

1.5 Most Important Style and Coding Rules

Here is a selection of C++ rules we have tended to follow. Detailed more below, example scripts are included to reformat the code style however you may prefer for local editing. However, if committing changes to the repository, only use the default formatting in the example scripts.

- snake_case is used for variable and class names.
- We use "one true brace" style (OTBS) for source files.
- In header files, all functions are inline style with attached brackets.
- To keep code compact horizontally, indent is set to two spaces. Namespaces are not indented.
- Generally, all if-else and for-loops have brackets even if a single line.
- Accessor functions are not prefixed, i.e. a function would be called `name()` not `get_name()`, except where conflicts require a prefix. Functions for modifying variables are prefixed with `set_`,
- We use .h and .cc instead of .hpp and .cpp
- As much implementation as possible should go in .cc files. Header files can end up in long dependency lists in the make system. Small changes to header files can result in long recompiles. If the function is more than a basic set/get, put it into a .cc file.
- Header files with many classes are discouraged. When reasonable, one class per header file/source file is preferred. Many short files are better than a few really long ones.
- Document, document, document. If it isn't obvious what a function does, add doxygen-compatible documentation. Examples are better than abstract wording.
- Use STL and Boost containers for data structures. Do not worry about performance until much later. Premature optimization is the root of all evil. If determined that an optimized data structure is needed, please do that after the entire class is complete and debugged.
- Forward declarations. There are a lot of interrelated classes, often creating circular dependencies. In addition, you can add up with an explosion of include files, often involving classes that are never used in a given `.cc` file. For cleanliness of compilation, you will find many `*_fwd.h` files throughout the code. If you need a forward declaration, we encourage including this header file rather than ad hoc forward declarations throughout the code.

Since we respect the sensitivity of code-style wars, we include scripts that demonstrate basic usage of the C++ code formatting tool *astyle*. This can be downloaded from http://astyle.sourceforge.net. A python script called `fix_style` is included in the top-level bin directory. It recursively reformats all files in a given directory and its subdirectories.

Chapter 2

SST/macro Classes

2.1 Factory Types

We here introduce factory types, i.e. polymorphic types linked to keywords in the input file. String parameters are linked to a lookup table, finding a factory that produces the desired type. In this way, the user can swap in and out C++ classes using just the input file. There are many distinct factory types relating to the different hardware components. There are factories for topology, NIC, node, memory, switch, routing algorithm - the list goes on. Here show how to declare a new factory type and implement various polymorphic instances. The example files can be found in `tutorials/programming/factories`.

2.1.1 Usage

Before looking at how to implement factory types, let's look at how they are used. Here we consider the example of an abstract interface called `actor`. The code example is found in `main.cc`. The file begins

```
1  #include <sstmac/skeleton.h>
2  #include "actor.h"
3
4  namespace sstmac {
5      namespace tutorial {
6
7  #define sstmac_app_name rob_reiner
8
9  int main(int argc, char **argv)
10 {
```

The details of declaring and using external apps is found in the user's manual. Briefly, SST/macro (using the `sstmac_app_name` define) reroutes the main function to be callable within a simulation. From here it should be apparent that we defined a new application with name `rob_reiner`. Inside the main function, we create an object of type `actor`.

```
1  actor* the_guy = actor_factory::get_param("actor_name", get_params());
2  the_guy->act();
3  return 0;
```

We use the `actor_factory` to create the object. The value of `actor_name` is read from the input file `parameters.ini` in the directory. Depending on the value in the input file, a different type will be created. The input file contains several parameters related to constructing a machine model - ignore these for now. The important parameters are:

```
node {
 app1 {
  name = rob_reiner
  biggest_fan = jeremy_wilke
  actor_name = patinkin
  sword_hand = right
 }
}
```

Using the Makefile in the directory, if we compile and run the resulting executable we get the output

```
Hello. My name is Inigo Montoya. You killed my father. Prepare to die!
Estimated total runtime of           0.00000000 seconds
SST/macro ran for        0.0025 seconds
```

If we change the parameters:

```
node {
 app1 {
  name = rob_reiner
  biggest_fan = jeremy_wilke
  actor_name = guest
  num_fingers = 6
 }
}
```

we now get the output

```
You've been chasing me your entire life only to fail now.
I think that's the worst thing I've ever heard. How marvelous.
Estimated total runtime of           0.00000000 seconds
SST/macro ran for        0.0025 seconds
```

Changing the values produces a different class type and different behavior. Thus we can manage polymorphic types by changing the input file.

2.1.2 Base Class

To declare a new factory type, you must include the factory header file

```
#include <sprockit/factories/factory.h>

namespace sstmac {
    namespace tutorial {

class actor {
```

We now define the public interface for the actor class

```
 public:
  actor(sprockit::sim_parameters* params);

  virtual void act() = 0;

  virtual ~actor(){}
```

Again, we must have a public, virtual destructor. Each instance of the actor class must implement the `act` method.

For factory types, each class must take a parameter object in the constructor. The parent class has a single member variable

```
 protected:
  std::string biggest_fan_;
```

After finishing the class, we need to invoke a macro

```
DeclareFactory(actor);
```

making SST/macro aware of the new factory type.

Moving to the `actor.cc` file, we see the implementation

```
namespace sstmac {
    namespace tutorial {

actor::actor(sprockit::sim_parameters* params)
{
  biggest_fan_ = params->get_param("biggest_fan");
}
```

We initialize the member variable from the parameter object. We additionally need a macro

```
ImplementFactory(sstmac::tutorial::actor);
```

that defines certain symbols needed for implementing the new factory type. For subtle reasons, this must be done in the global namespace.

2.1.3 Child Class

Let's now look at a fully implemented, complete actor type. We declare it

```
#include "actor.h"

namespace sstmac {
    namespace tutorial {

class mandy_patinkin :
    public actor
{
 public:
  mandy_patinkin(sprockit::sim_parameters* params);

  FactoryRegister("patinkin", actor, mandy_patinkin,
    "He's on one of those shows now... NCIS? CSI?");
```

We have a single member variable

```
 private:
  std::string sword_hand_;
```

This is a complete type that can be instantiated. To create the class we will need the constructor:

```
mandy_patinkin(sprockit::sim_parameters* params);
```

And finally, to satisfy the `actor` public interface, we need

```
virtual void act() override;
```

In the class declaration, we need to invoke the macro `FactoryRegister` to register the new child class type with the given string identifier. The first argument is the string descriptor that will be linked to the type. The second argument is the parent base class. The third argument is the specific child type. Finally, a documentation string should be given with a brief description. We can now implement the constructor:

```
mandy_patinkin::mandy_patinkin(sprockit::sim_parameters* params) :
  actor(params)
{
  sword_hand_ = params->get_param("sword_hand");

  if (sword_hand_ == "left"){
    sprockit::abort("I am not left handed!");
  }
  else if (sword_hand_ != "right"){
      spkt_abort_printf(value_error,
          "Invalid hand specified: %s",
          sword_hand_.c_str());
  }
}
```

The child class must invoke the parent class method. Finally, we specify the acting behavior

```
void mandy_patinkin::act()
{
    std::cout << "Hello. My name is Inigo Montoya. You killed my father. Prepare to die!"
              << std::endl;
}
```

Another example `guest.h` and `guest.cc` in the code folder shows the implementation for the second class.

Chapter 3

SST/macro Connectable Interface

3.1 Required Functions

Hardware components communicate via ports. Component 1 sends an event out on one port. Component 2 receive the event in on another port. During simulation setup, components must have their ports "connected" together. Creating a connection or link from an output port to an input port requires registering event handlers for each end of the link. The source component sends events out to a payload handler. Upon arriving at the destination component, the payload handler is invoked for that event. After receiving the event, the destination component can optionally send an ack or credit back to the source. Thus a link can also have credit handlers registered.

SST/macro actually provides a thin wrapper around the core SST interface. In SST/macro, ports are integers. In SST core, ports are labeled by strings. Similarly, payload and credit handlers are not automatically set up in SST core. SST/macro forces links and handlers to be set up automatically.

Every hardware component in SST/macro should inherit from `connectable_component` in `connection.h` There are four critical abstract functions in the virtual interface. First:

```
1  virtual void connect_output(
2    sprockit::sim_parameters* params,
3    int src_outport,
4    int dst_inport,
5    event_handler* payload_handler) = 0;
```

This is invoked on the source component of a link giving the port numbers on either end of the link. It gives the source component the payload handler that will be invoked on the destination component. The final complication here is the parameters object. The parameters passed in here are any port-specific parameters. These include all the default parameters for the port (that may not be port-specific) plus all parameters in the namespace `portN` for a given port number. Parameter namespaces are covered in the user's manual.

The next connection function is:

```
1  virtual void connect_input(
2    sprockit::sim_parameters* params,
3    int src_outport,
4    int dst_inport,
5    event_handler* credit_handler) = 0;
```

Similar to `connect_output`, this is invoked on the destination component of a link. Instead of giving a payload handler to receive new events, it receives a credit handler that the destination should send acks and credits to. The parameters work the same way as the output parameters.

But where do the handlers come from? Connectable objects must implement:

```
virtual link_handler* credit_handler(int port) const = 0;

virtual link_handler* payload_handler(int port) const = 0;
```

These `link_handler` objects are a special instance of `event_handler`. Each class must return the correct payload and credit handlers for each valid port. The handler and port will then be passed to the corresponding `connect_output` or `connect_input` function.

`link_handler` objects are created as functors for particular member functions of a class. They are created through the helper function:

```
template <class T, class Fxn>
link_handler* new_link_handler(const T* t, Fxn fxn){
  return new_handler<T,Fxn>(const_cast<T*>(t), fxn);
}
```

Given a class `Test` with a member function

```
void Test::handle_payload(event* ev)
```

we could create the appropriate `link_handler` as

```
link_handler* Test::payload_handler(int port) const {
  return new_link_handler(this, &Test::handle);
}
```

3.2 Example External Component

An example component source file, corresponding Makefile for generating the external library, and parameter file demonstrating its usage can be found in `skeletons/sst_component_example`. Some critical things to note from the file `component.cc` are the component registration macro and the Python module generation. The Python module generation is specific to SST core and is not part of SST/macro.

The component registration macro is:

```
RegisterComponent("dummy", test_component, dummy_switch,
       "test", COMPONENT_CATEGORY_NETWORK,
       "A dummy switch for teaching")
```

This is similar to the factory registration macro, but extends it for the special case of independent hardware components. If the component registration macro is used, then the factory registration macro is not required. The first field is a unique string identifier for the component. The second field is the parent factory type. The third field is the actual class name. The fourth field is the module name matching the Python module name at the top of the `component.cc` file. The fifth field is a generic component category. The currently allowed categories are defined in SST core as:

```
#define COMPONENT_CATEGORY_UNCATEGORIZED  0x00
#define COMPONENT_CATEGORY_PROCESSOR      0x01
#define COMPONENT_CATEGORY_MEMORY         0x02
#define COMPONENT_CATEGORY_NETWORK        0x04
#define COMPONENT_CATEGORY_SYSTEM         0x08
```

The final field is a documentation string.

All of the required connection functions are implemented in `component.cc`.

3.2.1 Python configuration

The Python file `run.py` in the same folder shows the simplest possible setup with two components connected by a single both on port 0. First, we import the necessary modules. The file `component.cc` implements a module called `test` that we load by calling `import sst.test`. We also load all Python functions provided by the macro library.

```
import sst
from sst.macro import *
import sst.test
```

We then make components, e.g.

```
latency="1us"
comp1 = sst.Component("1", "test.dummy_switch")
comp1.addParam("id", 1)
comp1.addParam("latency", latency)
```

and another component

```
comp2 = sst.Component("2", "test.dummy_switch")
comp2.addParam("id", 2)
comp2.addParam("latency", latency)
```

And finally connect them with a link using a SST/macro helper function:

```
port=0
comp1Id=1
comp2Id=2
makeBiNetworkLink(comp1,comp1Id,port,
                  comp2,comp2Id,port,
                  latency)
```

The code in the Python script causes `connect_output` and `connect_input` to be invoked on port 0 for each of the components.

3.2.2 Makefile

The Makefile uses compiler wrappers installed with SST/macro. These differ from the compiler wrappers used for skeleton applications discussed in the user's manual.

```
CXX :=      libsst++
CC :=       libsstcc
CXXFLAGS := -fPIC
```

All components should be compiled with `-fPIC` for use in shared library. Making generates a `libtest.so` that can be loaded using the Python setup or through the `external_libs` parameter in a `.ini` file.

Chapter 4

SProCKit

SST/macro is largely built on the Sandia Productivity C++ Toolkit (SProCKit), which is included in the SST/macro distribution. Projects developed within the simulator using SProCKit can easily move to running the application on real machines while still using the SProCKit infrastructure. One of the major contributions is reference counted pointer types. The parameter files and input deck are also part of SProCKit.

4.1 Debug

The goal of the SProCKit debug framework is to be both lightweight and flexible. The basic problem encountered in SST/macro development early on was the desire to have very fine-grained control over when and where something prints. Previously declared debug flags are passed through the `debug_printf` macro.

```
1 debug_printf(sprockit::dbg::mpi,
2   "I am MPI rank %d of %d",
3   rank, nproc);
```

The macro checks if the given debug flag is active. If so, it executes a `printf` with the given string and arguments. Debug flags are turned on/off via static calls to

```
1 sprockit::debug::turn_on(sprockit::dbg::mpi);
```

SST/macro automatically turns on the appropriate debug flags based on the `-d` command line flag or the `debug =` parameter in the input file.

Multiple debug flags can be specified via OR statements to activate a print statement through multiple different flags.

```
1 using namespace sprockit;
2 debug_printf(dbg::mpi | dbg::job_launch,
3   "I am MPI rank %d of %d",
4   rank, nproc);
```

Now the print statement is active if either MPI or job launching is going to be debugged.

In `sprockit/debug.h` a set of macros are defined to facilitate the declaration. To create new debug flags, there are two macros. The first, `DeclareDebugSlot`, generally goes in the header file to make the flag visible to all files. The second, `RegisterDebugSlot`, goes in a source file and creates the symbols and linkage for the flag.

```
launch.h:
DeclareDebugSlot(job_launch);

launch.cc
RegisterDebugSlot(job_launch);
```

4.2 Serialization

Internally, SST/macro makes heavy use of object serialization/deserialization in order to run in parallel with MPI. To create a serialization archive, the code is illustrated below. Suppose we have a set of variables

```
struct point {
 int x;
 int y;
}
point pt;
pt.x = 0;
pt.y = 2;
int niter = 5;
std::string str = "hello";
```

We can serialize them to a buffer

```
sstmac::serializer ser;
ser.set_mode(sstmac::serializer::PACK);
ser.init(new char[512]);
ser & pt;
ser & niter;
ser & str;
```

In the current implementation, the buffer must be explicitly given.

To reverse the process for a buffer received over MPI, the code would be

```
char* buf = new char[512];
MPI_Recv(buf, ...)
sstmac::serializer;
ser.set_mode(sstmac::serializer::UNPACK);
ser.init(buf);
ser & pt;
ser & niter;
ser & str;
```

Thus the code for serializing is exactly the same as deserializing. The only change is the mode of the serializer is toggled. The above code assumes a known buffer size (or buffer of sufficient size). To serialize unknown sizes, the serializer can also compute the total size first.

```
sstmac::serializer ser;
ser.reset();
ser.set_mode(sstmac::serializer::SIZER);
ser & pt;
ser & niter;
ser & str;
int size = ser.sizer.size(); //would be 17 for example above
char* buf = new char[size];
...
```

The known size can now be used safely in serialization.

The above code only applies to plain-old dataypes and strings. The serializer also automatically handles STL containers through the `&` syntax. To serialize custom objects, a C++ class must implement the serializable interface.

```
namespace my_ns {
class my_object :
  public sstmac::serializable
{
 ImplementSerializable(my_object)
 ...
 void serialize_order(sstmac::serializer& ser);
 ...
};
}
```

The serialization interface requires inheritance from `serializable`. This inheritance forces the object to define a `serialize_order` function. The macro `ImplementSerializable` inside the class creates a set of necessary functions. This is essentially a more efficient RTTI, mapping unique integers to a polymorphic type. The forced inheritance allows more safety checks to ensure types are being set up and used correctly. All that remains now is defining the `serialize_order` in the source file:

```
void my_object::serialize_order(sstmac::serializer& ser)
{
  ser & my_int_;
  set << my_double_;
  ...
}
```

For inheritance, only the top-level parent class needs to inherit from `serializable`.

```
class parent_object :
  public sstmac::serializable
{
...
  void serialize_order(sstmac::serializer& set);
...
};

class my_object :
  public parent_object
{
 ImplementSerializable(my_object)
 ...
 void serialize_order(sstmac::serializer& ser);
 ...
};
```

In the above code, only `my_object` can be serialized. The `parent_object` is not a full serializable type because no descriptor is registered for it using the macro `ImplementSerializable`. Only the child can be

serialized and deserialized. However, the parent class can still contribute variables to the serialization. In the source file, we would have

```
void my_object::serializer_order(sstmac::serializer& ser)
{
  parent_object::serialize_order(ser);
  ...
}
```

The child object should always remember to invoke the parent serialization method.

4.3 Keyword Registration

As stated previously, SProCKit actually implements all the machinery for parameter files. This is not part of the SST/macro core. To avoid annoying bugs, the SProCKit input system provides mechanisms for having allowed values be declared ahead of time. This can happen in any source file through static initialization macros. Only one invocation is allowed per source file, but keywords can be registered in as many source files as desired. The macro is used in the global namespace:

```
RegisterKeywords("nx", "ny", "nz");
```

This registers some basic keywords that might be used in a 3D grid application. If strict mode is turned on, any parameters in the input file not matching a known parameter will produce an error.

In many cases a parameter is an enumerated value or fits a pattern. SProCKit allows regular expressions to be declared as valid patterns for a keyword.

```
StaticKeywordRegisterRegexp my_regexp("particle\\d+");
```

Here you create a static instance of a keyword registration object. The constructor registers the regular expression. Now, any keywords matching the regular expression will be considered valid.

Chapter 5

Discrete Event Simulation

There are abundant tutorials on discrete event simulation around the web. To understand the basic control flow of SST/macro simulations, you should consult Section 3.6, Discrete Event Simulation, in the user's manual. For here, it suffices to simply understand that objects schedule events to run at a specific time. When an event runs, it can create new events in the future. A simulation driver gradually progresses time, running events when their time stamp is reached. As discussed in the user's manual, we must be careful in the vocabulary. *Simulation time* or *simulated time* is the predicted time discrete events are happening in the simulated hardware. *Wall time* or wall clock time is the time SST/macro itself has been running. There are a variety of classes the cooperate in driving the simulation, which we now describe.

5.1 Event Managers

The driver for simulations is an event manager that provides the function

```
1 virtual void schedule(timestamp start_time, event* event) = 0;
```

This function must receive events and order them according to timestamp. Two main types of data structures can be used, which we briefly describe below.

The event manager also needs to provide

```
1 virtual void run() = 0;
```

The termination condition can be:

- A termination timestamp is passed. For example, a simulation might be specified to terminate after 100 simulation seconds. Any pending events after 100 seconds are ignored and the simulation exits.
- The simulation runs out of events. With nothing left to do, the simulation exits.

Events are stored in a queue (sorted by time)

```
namespace sstmac {

class event_queue_entry
{
 public:
  virtual void execute() = 0;

  ...
};
```

The execute function is invoked by the `event_manager` to run the underlying event. There are generally two basic event types in SST/macro, which we now introduce.

5.1.1 Event Handlers

In most cases, the event is represented as an event sent to an object called an `event_handler` at a specific simulation time. In handling the event, the event handlers change their internal state and may cause more events by scheduling new events at other event handlers (or scheduling messages to itself) at a future time. The workhorses for SST/macro are therefore classes that inherit from `event_handler`. The only method that must be implemented is

```
void handle(event* ev);
```

The function is the common interface for numerous different operations from network injection to memory access to MPI operations. In general, objects have two "directions" for the action - send or receive. A NIC could "handle" a packet by injecting it into the network or "handle" a message by reporting up the network stack the message has arrived. In most cases, the handled message must therefore carry with it some notion of directionality or final destination. An event handler will therefore either process the message and delete the message, or, if that handler is not the final destination, forward it along. Some event handlers will only ever receive, such as a handler representing a blocking `MPI_Recv` call. Some event handlers will always receive and then send, such as network switches who are always intermediate steps between the start and endpoints of a network message.

In most cases, events are created by calling the function

```
void schedule(const timestamp &t,
  event_handler* handler,
  event* ev);
```

This then creates a class of type `event_queue_entry`, for which the execute function is

```
void handler_event_queue_entry::execute()
{
  if (!canceled_) {
    handler_->handle(ev_to_deliver_);
  }
}
```

Objects can inherit from `event_handler` to create new event handlers. Preferred usage, though, is on-the-fly creation of event handlers through C++ templates. The interface does not actually expose C++ templates. The function `new_handler` defined in `event_callback.h` has the prototype:

```
template<class Cls, typename Fxn, class... Args>
event_handler*
new_handler(Cls* cls, Fxn fxn, const Args&... args);
```

Here `Fxn` is a member function pointer. When an `event* ev` is scheduled to the event handler, the member function pointer gets invoked:

```
(cls->*fxn)(ev, args...);
```

For example, given a class `actor` with the member function `act`

```
void actor::actor(event* ev, int ev_id){...}
```

we can create an event handler

```
actor* a = ....;
event_handler* handler = new_handler(a, &actor::act, 42);
...
event* ev = ....;
schedule(time, handler, ev);
```

When the time arrives for the event, the member function will be invoked

```
a->act(ev, 42);
```

5.1.2 Event Heap/Map

The major distinction between different event containers is the data structured used. The simplest data structure is an event heap or ordered event map. The event manager needs to always be processing the minimum event time, which maps naturally onto a min-heap structure. Insertion and removal are therefore log(N) operations where N is the number of currently scheduled events. For most cases, the number and length of events is such that the min-heap is fine.

5.2 Event Schedulers

The simulation is partitioned into objects that are capable of scheduling events. Common examples of `event_scheduler` objects are nodes, NICs, memory systems, or the operating system. In serial runs, an event scheduler is essentially just a wrapper for the `event_manager` and the class is not strictly necessary. There are two types of event scheduler: `event_component` and `event_subcomponent`. In parallel simulation, though, the simulation must be partitioned into different scheduling units. Scheduling units are then distributed amongst the parallel processes. Components are the basic unit. Currently only nodes and network switches are components. All other devices (NIC, memory, OS) are subcomponents that must be linked to a parent component. Even though components and subcomponents can both schedule events (both inherit from `event_scheduler`), all subcomponents must belong to a component. A subcomponent cannot be separated from its parent component during parallel simulation.

Chapter 6

Software Models

The driver for most simulations is a skeleton application. Although this can be arbitrary source code, we will consider the example of an MPI application below. We will discuss distributed services in Section 6.3 below, which is similar to an application. In general, when we refer to applications we mean scientific codes or client codes that are doing "domain-specific" work. These will be different from service applications like parallel file systems.

We will be very specific with the use of the terms "virtual" and "real" or "physical". Virtual refers to anything being modeled in the simulator. Real or physical refers to actual processes running on a supercomputer. When referring to a skeleton application in the simulator, we refer only to virtual MPI *ranks*. The term *process* only applies to physical MPI ranks since virtual MPI ranks are not true processes, but rather a user-space thread. Many virtual MPI ranks can be colocated within the same process. A physical process generally has:

- A heap
- A stack
- A data segment (global variable storage)

As much as possible, SST/macro strives to make writing skeleton app source code the same as writing source for an actual production application. All virtual MPI ranks can share the same heap within a process. There is no strict requirement that each virtual MPI rank have its own contiguous heap. Rather than make each MPI stack a full pthread, each MPI rank is created as a lightweight user-space thread. In the simulation, all virtual MPI ranks will be running "concurrently" but not necessarily in "parallel." Each MPI rank within a process must time-share the process, meaning the simulator core will context switch between each MPI stack to gradually make forward progress. While the heap and stack are easy to provide for each virtual MPI rank, there is no easy way to provide a unique global variable segment to each MPI rank. This therefore requires the Clang source-to-source compiler to redirect all global variable accesses to user-space thread specific locations.

6.1 Applications and User-space Threads

In a standard discrete event simulation (DES), you have only components and events. Components send events to other components, which may in turn create and send more events. Each event has an arrival time associated with it. Components must process events in time-order and keep a sorted queue of events. Time advances in the simulation as components pop off and process events from the queue.

SST/macro mixes two modes of advancing time. The simulator has node, NIC, router, and memory models that advance "hardware time" in the standard way. Applications are not a regular DES component. They are just a stack and a heap, without an event queue or `handle(event* ev)` callback functions. Applications just step through instructions, executing to the end of the application. To advance time, applications must *block* and *unblock*. In DES, time does not advance *during* events - only between events. For applications, time does not advance while the application is executing. It advances between a block/unblock pair. This is clearly somewhat counterintuitive (time advances while an application is not executing).

The details of block/unblock should never be apparent in the skeleton code. A skeleton MPI code for SST/macro should *look* exactly like a real MPI code. To advance time, certain function calls must be intercepted by SST/macro. This is done through a combination of compile-time/linker-time tricks. All skeleton apps should be compiled with the `sst++` compiler wrapper installed by SST/macro. The compiler wrapper sets include paths, includes headers, and directs linkage.

For MPI, this first occurs through include paths. SST/macro installs an `mpi.h` header file. Thus compiling with `sst++` includes a "virtual" MPI header designed for SST. Next, the MPI header defines macros that redirect MPI calls.

```
1 ...
2 #define MPI_Send(...) sumi::sstmac_mpi()->send(__VA_ARGS__)
3 ...
```

`sstmac_mpi` is a function we will explore more below. Inside `MPI_Send`, the SST/macro core takes over. When necessary, SST/macro will block the active user-space thread and context-switch.

We can illustrate time advancing with a simple `MPI_Send` example. We have discussed that a user-space thread is allocated for each virtual MPI rank. The discrete event core, however, still runs on the main application thread (stack). Generally, the main thread (DES thread) will handle hardware events while the user-space threads will handle software events (this is relaxed in some places for optimization purposes). Figure 6.1, shows a flow chart for execution of the send. Operations occurring on the application user-space thread are shaded in blue while operations on the DES thread are shaded in pink. Function calls do not advance time (shown in black), but scheduling events (shown in green) do advance time. Again, this is just the nature of discrete event simulation. The dashed edge shows a block/unblock pair. The call to `mpi_api::send` blocks after enqueuing the send operation with the OS. Virtual time in the simulation therefore advances inside the `MPI_Send` call, but the details of how this happens are not apparent in the skeleton app.

When the ACK arrives back from the NIC, the ACK signals to MPI that the operation is complete allowing it to unblock. The ACK is handled by a `service` object (which is the SST-specific implementation of an MPI server). A `service` is a special type of object, that we will discuss in more detail below.

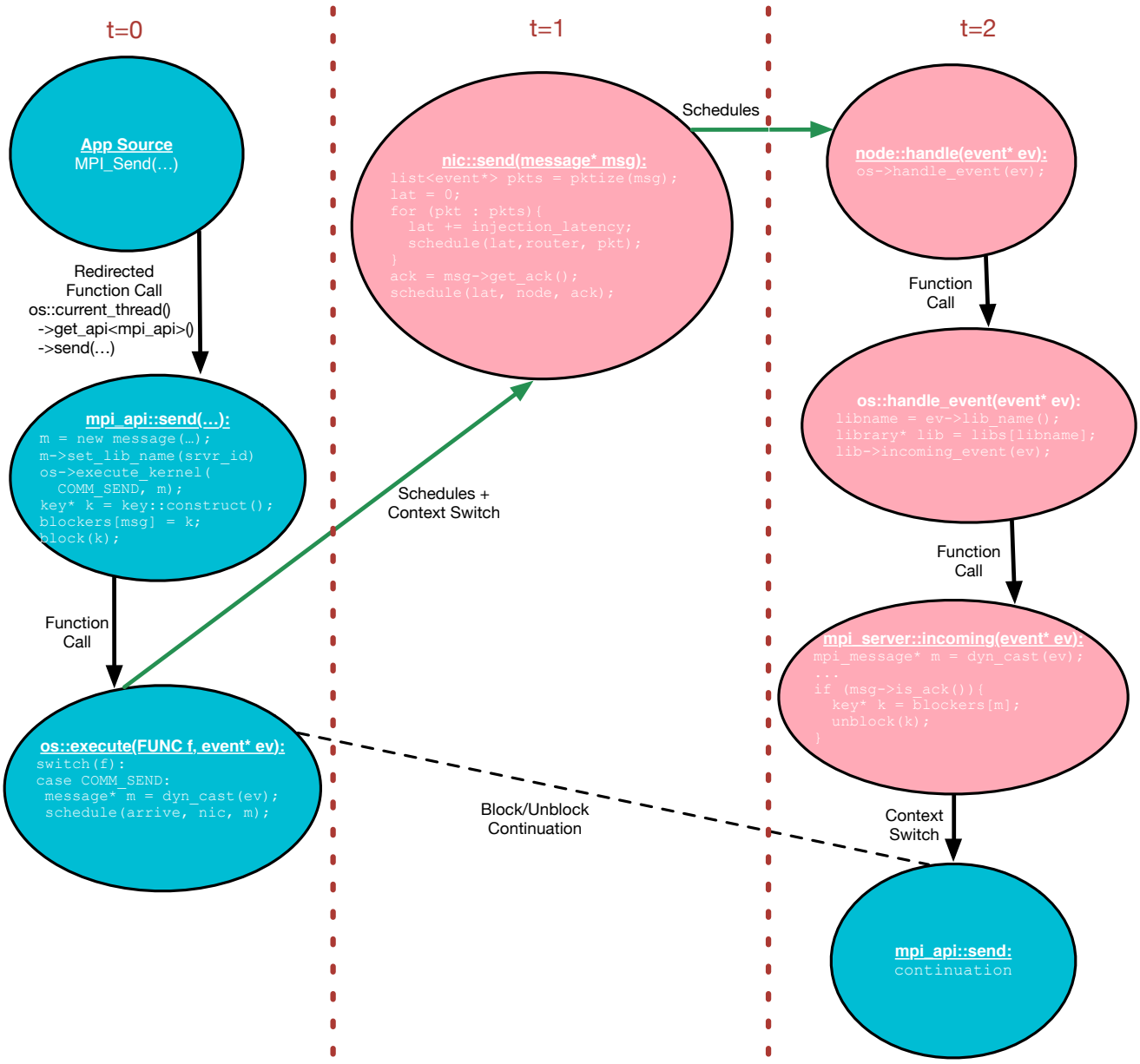


Figure 6.1: Flow of events for completing a send operation. Shows basic function calls, block/unblock context switches, and event schedules. User-space thread (application) operations are shown in blue. Main event thread (OS/kernel) operations are shown in pink.

	OS	Node	API	Service
Runs on Thread	Both user-space and main DES thread	Only main DES thread (user-space with rare exceptions for optimization)	Only user-space thread	Only main DES thread
How Advances Time	Both blocking and scheduling events, depending on context	Scheduling events to other components	Blocking or unblocking	Scheduling events to other components
Receives Events Via	Function calls from API/Service or from Node via `handle_event` function	Function calls from OS, receives scheduled events via the `handle` function	Does not usually receive events - only blocks or unblocks	OS forwards messages to `incoming_event` function
Sends Events Via	Makes function calls and schedules events to Node	Makes function calls and schedules events to NIC, Memory	Does not usually send events - only blocks or unblocks	Makes function calls and schedules events to OS, unblocks APIs

6.1.1 Thread-specific storage

Let us look at the capture of the `MPI_Send` call. First, a macro redefines the call to avoid symbol clashes if using an MPI compiler (both virtual and real MPI cannot share symbols). MPI uses global function calls to execute, meaning MPI has to operate on global variables. However, SST/macro cannot use global variables. Thus all state specific to each MPI rank (virtual thread) must be stashed somewhere unique. This basically means converting all global variables into user-space thread-local variables. When each user-space thread is spawned a `thread` object is allocated and associated with the thread. This object acts as a container for holding all state specific to that virtual thread. Rather than store MPI state in global variables, MPI state in stored in a class `mpi_api`, with one `mpi_api` object allocated per virtual MPI rank. The operating system class provides a static function `current_thread` that makes the leap from global variables to thread-local storage. To access a specific API, a special helper template function `get_api` exists on each thread object. Thus, instead of calling a global function `MPI_Send`, SST/macro redirects to a member function `send` on an `mpi_api` object that is specific to a virtual MPI rank.

6.2 Libraries

The creating and handling of software events is managed through `library` objects. Each `node` has an `operating_system` member variable that will manage software events on a virtual compute node. Each library object is stored in a lookup table in the operating system, allowing the operating system to access specific libraries at specific times.

```
class operating_system {
  ...
  map<string,library*> libs_;
  ...
};
```

Each library object has access to the operating system, being allowed to call

```
event* ev = ...
os_->execute_kernel(ty, ev);
```

where an enum argument `ty` specifies the type of computation or communication and the event object carries all the data needed to model it. This allows a library to *call out* to the operating system. Calls to `execute` *always* occur on a user-space thread. In executing a computation or communication, the operating system may block inside *execute* to advance time to complete the operation. Calls to `execute_kernel` can occur on either a user-space thread or the main event thread. Here the operating system acts as a service and *never* blocks.

Conversely, each `library` must provide a `incoming_event` method that allows the operating system to call back to the library

```
void incoming_event(event* ev){...}
```

Generally speaking, event notifications will arrive from the NIC (new messages, ACKs), memory system (data arrived), processor (computation complete), etc. These hardware events must be routed to the correct software library for processing.

The operating system provides an abstract interface (independent of the exact implementation of threading library) through `key` objects. Each blocking call:

```
key* k = key::construct();
os->block(k);
```

must be matched with a corresponding unblock on the same key object. Usually libraries stash keys in some sort of lookup table to track event completions.

```
key* k = blockers[ev->uid()];
os->unblock(k);
```

```
void operating_system::handle_event(event* ev) {
  library* lib = libs_[ev->lib_name()];
  lib->incoming_event(ev);
}
```

In order to route events to the correct library, the operating system maintains a string lookup table of `library` objects. All events associated with that library must be constructed with the correct string label, accessible through the event accessor function `lib_name`.

6.2.1 API

The SST/macro definition of API was alluded to in 6.1.1. The base `api` class inherits from `library`. All API code must execute on a user-space thread. API calls are always associated with a specific virtual

MPI rank. To advance time, API calls must block and unblock. Functions executing on user-space threads are "heavyweight" in the sense that they consume resources. API compute calls must allocate cores via a compute scheduler to execute.

API objects are accessible in skeleton apps through a global template function is provided in `sstmac/skeleton.h`.

```
template <class T>
T* get_api();
```

for which the implementation is

```
thread* thr = operating_system::current_thread();
return thr->get_api<T>();
```

which converts the global template function into a thread-specific accessor. The most prominent example of an API is the `mpi_api` object for encapsulating an MPI rank. Other prominent examples include the various computation APIs such as `blas_api` that provides bindings for simulation various linear algebra functions.

6.2.2 Service

In contrast to an API that always executes on a user-space, services always execute on the main event thread. They therefore never block (or unblock). Services do not perform compute-heavy operations and therefore, in a simulator approximation, do not consume resources. For software to advance time, it must block and then unblock after a virtual time delay. Services therefore have no ability to advance time and therefore all service operations are "instantaneous." Again, this is an approximation for the sake of simulation simplicity and efficiency. While an API can have several instances on a node (e.g. several MPI ranks on the same compute node), services are generally unique. Examples of services are MPI or filesystem services that sort messages arriving from the NIC and pass them off to the correct API or library object. If a service is "heavy-weight" and must model computational delays, it must run as a full thread with its own user-space thread stack.

6.3 Distributed Service

In many cases, a service is just local to a node and is part of the operating system or a runtime library, such as a local filesystem or a server receiving incoming network messages. In cases of something like a distributed file system like LUSTRE or distributed key-value store like Memcached, a service spans multiple nodes. These services are like skeleton applications in that they require a parallel launch with node allocation and rank indexing.

To create a new distributed service, you have to inherit from `sstmac::distributed_service` in `sstmac/libraries/sumi/distributed_service.h`. Distributed services require a network transport layer to communicate between service nodes. In contrast to other libraries and services that are found in `sstmac/software`, distributed services are linked to SUMI (SST unified message interface). A distributed service must implement the `void run()` virtual member function. In general, the `run` function will almost always perform something like:

```
void example_service::run()
{
 //some init
 bool block  = true;
```

```
 message* msg = poll_for_message(block);
 while (msg && !terminated()){
   process(msg);
   msg = poll_for_message(block);
 }
}
```

The function `poll_for_message` goes into a network polling loop looking for incoming messages. Depending on the boolean parameter, the function will block until a message arrives or returns immediately if no message is available. The service may receive a terminate message. This gets processed automatically. Whether a shutdown message has been received can be queried for by the `terminated` function.

Launching a distributed service is very similar to launching an application with a few subtle differences. The services to be launched are listed in the input file as:

```
services = insitu_viz filesystem
```

As a space-separated list. This would launch services registered using the standard sprockit registration:

```
SpktRegister("filesystem", distributed_service, test_filesystem);
```

The launch parameters and any service-specific parameters are done exactly as for applications, but go in a namespace corresponding to the service:

```
filesystem.launch_cmd = aprun -n 10 -N 1
filesystem.allocation = first_available
filesystem.reed_solomon_k = 5
filesystem.reed_solomon_n = 7
...
```

This launches ten filesystem service nodes using a given node allocation strategy. Some filesystem-specific parameters are given (in this case dealing with Reed-Solomon protection codes).

6.3.1 Coordinating servers and clients

Slightly more complicated is the process of communicating between service and client applications. In many HPC simulations, the only thing being simulated is a single MPI application. The default `send` function in `sumi_transport` assumes messages are being sent *within* a given application.

```
void send_payload(int dst, message* msg, bool needs_ack);
```

The only thing that must be specified is a destination *rank*, which is the virtual ID within a given session layer. The transport layer itself will map that to the correct physical node address and deliver the message.

An alternative function exists for sending from client to server:

```
void client_server_send(
  int dest_rank,
  node_id dest_node,
  app_id dest_app,
  sumi::message* msg);
```

While a session rank must be given, a physical node ID and an application ID must be given. These can no longer be filled in automatically.

For a client to get all of the information about a distributed service, a special object is supplied that can be fetched:

```
sstmac::sw::app_launch* srv = sstmac::sw::app_launch::service_info("filesystem");
```

The object `srv` has functions that can be queried to figure out all the `client_server_send` parameters. The parameter to the static `service_info` function is the service name given to `SpktRegister` and given in the input file.

The same `client_server_send` function is used for both clients making requests to servers and servers returning responses to clients. In most cases, the clients will use `service_info` to "discover" the servers. In order to receive responses, the clients should send messages containing all the relevant info with rank, physical node ID, and app ID.

Generally, an application knowns when it is done (e.g. reaches some convergence criterion). A service does not usually have a clean termination condition since it never knows when more client requests may arrive. In this case, the function `shutdown_server` can be called by a client application.

```
void shutdown_server(
  int dest_rank,
  node_id dest_node,
  app_id dest_app);
```

A single shutdown message to any server is sufficient to bring down the entire service. The shutdown message will be automatically broadcast to all other nodes.

Chapter 7

Hardware Models

7.1 Overview

To better understand how hardware models are put together for simulating interconnects, we should try to understand the basic flow of event in SST/macro involved in sending a message between two network endpoints. We have already seen in skeleton applications in previous sections how an application-level call to a function like `MPI_Send` is mapping to an operating system function and finally a hardware-level injection of the message flow. Overall, the following steps are required:

- Start message flow with app-level function call
- Push message onto NIC for send
- NIC packetizes message and pushes packets on injection switch
- Packets are routed and traverse the network
- Packets arrive at destination NIC and are reassembled (potentially out-of-order)
- Message flow is pushed up network software stack

Through the network, packets must move through buffers (waiting for credits) and arbitrate for bandwidth through the switch crossbar and then through the ser/des link on the switch output buffers. The control-flow diagram for transporting a flow from one endpoint to another via packets is shown in Figure 7.1

We can dive in deeper to the operations that occur on an individual component, mostly important the crossbar on the network switch. Figure 7.2 shows code and program flow for a packet arriving at a network switch. The packet is routed (virtual function, configurable via input file parameters), credits are allocated to the packet, and finally the packet is arbitrated across the crossbar. After arbitration, a statistics callback can be invoked to collect any performance metrics of interest (congestion, traffic, idle time).

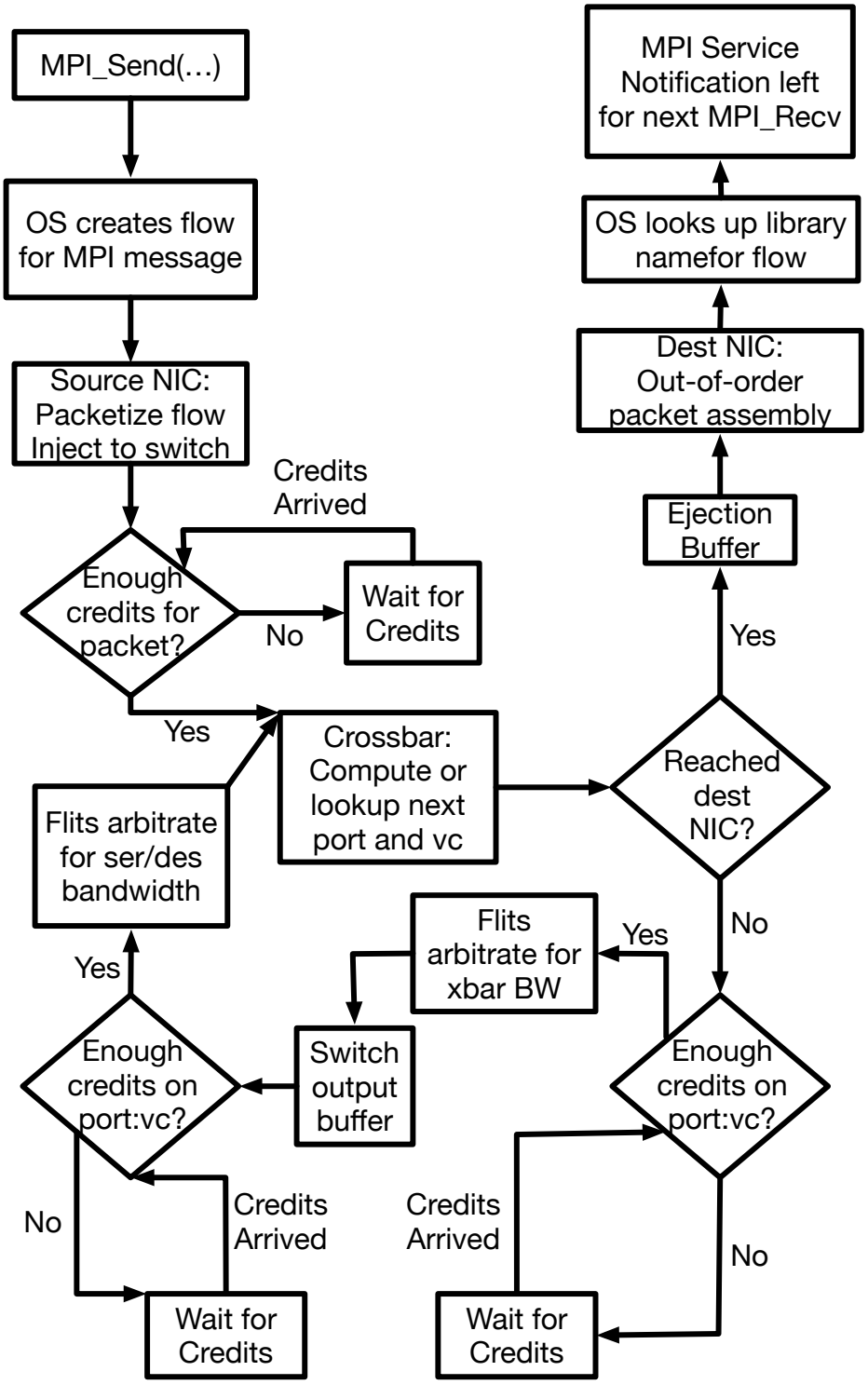


Figure 7.1: Decision diagram showing the various control flow operations that occur as a message is transport across the network via individual packet operations.

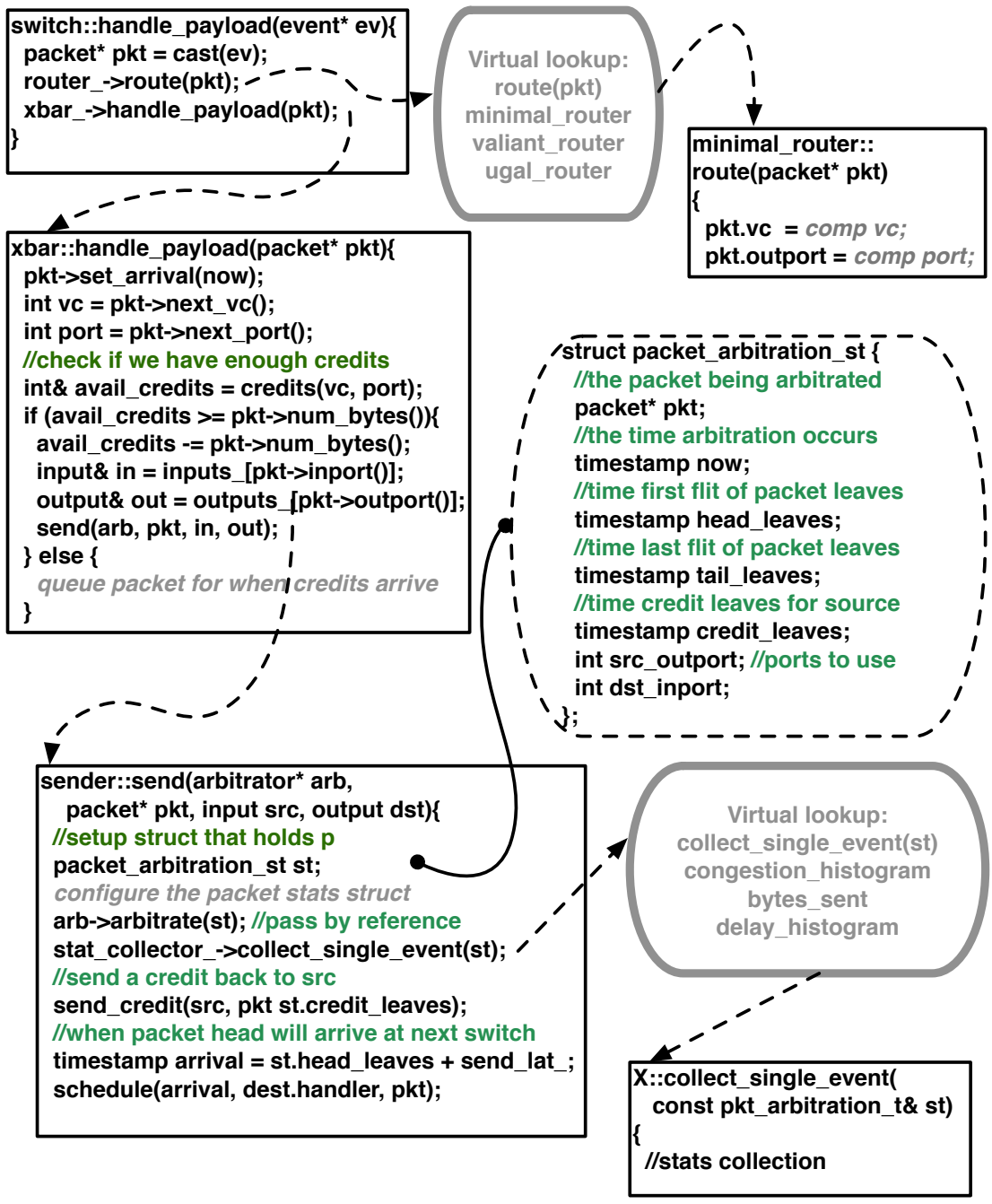


Figure 7.2: Code flow for routing, arbitration, and stats collections of packets traversing the crossbar on the network switch.

7.2 Connectables

With a basic overview of how the simulation proceeds, we can now look at the actual SST/macro class types. While in common usage, SST/macro follows a well-defined machine model (see below), it generally allows any set of components to be connected. As discussed in Chapter 5, the simulation proceeds by having event components exchange messages, each scheduled to arrive at a specific time. SST/macro provides a generic interface for any set of hardware components to be linked together. Any hardware component that connects to other components and exchanges messages must inherit from the `connectable` class. The `connectable` class presents a standard virtual interface

```
class connectable
{
 public:
  virtual void connect_output(
    sprockit::sim_parameters* params,
    int src_outport,
    int dst_inport,
    event_handler* handler) = 0;

  virtual void
  connect_input(
    sprockit::sim_parameters* params,
    int src_outport,
    int dst_inport,
    event_handler* handler) = 0;


};
```

First, port numbers must be assigned identifying the output port at the source and in the input port at the destination. For example, a switch may have several outgoing connections, each of which must be assigned a unique port number. The connection must be configured at both source and destination. The function is called twice for each side of the connection. If we have a source and destination:

```
connectable* src = ...
connectable* dst = ...
sprockit::sim_parameters* params = ...
src->connect_output(params, inport, outport, Output, dst);
dst->connect_input(params, inport, outport, Input, src);
```

A certain style and set of rules is recommended for all connectables. If these rules are ignored, setting up connections can quicky become confusing and produce difficult to maintain code. The first and most important rule is that `connectables` never make their own connections. Some "meta"-object should create connections between objects. In general, this work is left to a `interconnect` object. An object should never be responsible for knowing about the "world" outside itself. A topology or interconnect tells the object to make a connection rather than the object deciding to make the connection itself. This will be illustrated below in 7.8.

The second rule to follow is that a connect function should never call another connect function. In general, a single call to a connect function should create a single link. If connect functions start calling other connect functions, you can end up a with a recursive mess. If you need a bidirectional link (A $\rightarrow$ B, B $\rightarrow$ A), two separate function calls should be made

```
A->connect_output(B);
B->connect_input(A);
```

rather than having, e.g. A create a bidirectional link.

The first two rules should be considered rigorous. A third recommended rule is that all port numbers should be non-negative, and, in most cases, should start numbering from zero.

Combining the factory system for polymorphic types and the connectable system for building arbitrary machine links and topologies, SST/macro provides flexibility for building essentially any machine model you want. However, SST/macro provides a recommended machine structure to guide constructing machine models.

7.3 Interconnect

For all standard runs, the entire hardware model is driven by the interconnect object. The interconnect creates nodes, creates network switches, chooses a topology, and connects all of the network endpoints together. In this regard, the interconnect also choose what types of components are being connected together. For example, if you were going to introduce some custom FPGA device that connects to the nodes to perform filesystem operations, the interconnect is responsible for creating it.

To illustrate, here is the code for the interconnect that creates the node objects. The interconnect is itself a factory object, configured from a parameter file.

```
interconnect::interconnect(sprockit::sim_parameters* params, event_manager* mgr,
  partition* part, parallel_runtime* rt)
{
  sprockit::sim_parameters* top_params = params->get_namespace("topology");
  topology_ = topology_factory::get_param("name", top_params);
  num_nodes_ = topology_->num_nodes();
  num_switches_ = topology_->num_switches();

  switches_.resize(num_switches_);
  nodes_.resize(num_nodes_);

  sprockit::sim_parameters* node_params = params->get_namespace("node");
  sprockit::sim_parameters* switch_params = params->get_namespace("switch");

  sprockit::sim_parameters* nic_params = node_params->get_namespace("nic");
  sprockit::sim_parameters* inj_params = nic_params->get_namespace("injection");
  sprockit::sim_parameters* ej_params = switch_params->get_namespace("ejection");

  build_endpoints(node_params, nic_params, mgr);
  build_switches(switch_params, mgr);
  connect_endpoints(inj_params, ej_params);
  connect_switches(switch_params);
}
```

For full details of the functions that build/connect endpoints and switches, consult the source code. In general, the `interconnect` object uses the connectable interface to setup all the connections. It uses the topology interface to determine which connections are required, e.g.

```
switch_id src = ...
std::vector<topology::connection> outports;
topology_->connected_outports(src, outports);
for (auto& conn : outports){
  network_switch* dst_sw = switches_[conn.dst];
  src_sw->connect_output(params, conn.src_outport, conn.dst_inport,
            dst_sw->payload_handler(conn.dst_inport));
  dst_sw->connect_input(params, conn.src_outport, conn.dst_inport,
                src_sw->credit_handler(conn.src_outport));
}
```

The `connected_outports` function takes a given source switch and returns all the connections that the switch is supposed to make. Each switch must provide `payload_handler` and `ack_handler` functions to return the `event_handler` that should receive either new packets (payload) or credits (ack) for the connections.

7.4 Node

Although the `node` can be implemented as a very complex model, it fundamentally only requires a single set of functions to meet the public interface. The `node` must provide `execute_kernel` functions that are invoked by the `operating_system` or other other software objects. The prototypes for these are:

```
1 virtual void execute(ami::COMP_FUNC func, event* data);
2
3 virtual void execute(ami::SERVICE_FUNC func, event* data);
```

By default, the abstract `node` class throws an `sprockit::unimplemented_error`. These functions are not pure virtual. A node is only required to implement those functions that it needs to do. The various function parameters are enums for the different operations a node may perform: computation or communication. Computation functions are those that require compute resources. Service functions are special functions that run in the background and "lightweight" such that any modeling of processor allocation should be avoided. Service functions are run "for free" with no compute

7.5 Network Interface (NIC)

The network interface can implement many services, but the basic public interface requires the NIC to do three things:

- Inject messages into the network
- Receive messages ejected from the network
- Deliver ACKs (acknowledgments) of message delivery

For sending messages, the NIC must implement

```
1   virtual void do_send(network_message* payload);
```

A non-virtual, top-level `send` function performs operations standard to all NICs. Once these operations are complete, the NIC invokes `do_send` to perform model-specific send operations. The NIC should only ever send `network_message` types.

For the bare-bones class `logp_nic`, the function is

```
1 void logp_nic::do_send(network_message* msg)
2 {
3   long num_bytes = msg->byte_length();
4   timestamp now_ = now();
5   timestamp start_send = now_ > next_free_ ? now_ : next_free_;
6   timestamp time_to_inject = inj_lat_ + timestamp(inj_bw_inverse_ * num_bytes);
7   //leave the injection latency term to the interconnect
```

```
  schedule_now(injection_switch_, msg);

  next_free_ = start_send + time_to_inject;
  if (msg->needs_ack()) {
    //do whatever you need to do so that this msg decouples all pointers
    network_message* acker = msg->clone_injection_ack();
    schedule(next_free_, parent_->self_handler(), acker); //send to node
  }
}
```

After injecting, the NIC creates an ACK and delivers the notification to the `node`. In general, all arriving messages or ACKs should be delivered to the node. The node is responsible for generating any software events in the OS.

For receiving, messages can be moved across the network and delivered in two different ways: either at the byte-transfer layer (BTL) or message-transfer layer (MTL). Depending on the congestion model, a large message (say a 1 MB MPI message) might be broken up into many packets. These message chunks are moved across the network independently and then reassembled at the receiving end. Alternatively, for flow models or simple analytical models, the message is not packetized and instead delivered as a single whole. The methods are not pure virtual. Depending on the congestion model, the NIC might only implement chunk receives or whole receives. Upon receipt, just as for ACKs, the NIC should deliver the message to the node to interpret. In general, `nic::handle` is intended to handle packets. If a NIC supports direct handling of complete messages (MTL) instead of packets (BTL), it should provide a message handler:

```
event_handler* mtl_handler() const {
  return mtl_handler_;
}
```

A special completion queue object tracks chunks and processes out-of-order arrivals, notifying the NIC when the entire message is done.

7.6 Memory Model

As with the NIC and node, the memory model class can have a complex implementation under the hood, but it must funnel things through the a common function.

```
virtual void access(long bytes, double max_bw) = 0;
```

This function is intended to be called from an application user-space thread. As such, it should block until complete. For more details on the use of user-space threading to model applications, see the User's manual.

7.7 Network Switch

Unlike the other classes above, a network switch is not required to implement any specific functions. It is only required to be an `event_handler`, providing the usual `handle(event* ev)`. The internal details can essentially be arbitrary. However, the basic scheme for most switches follows the code below for the `pisces` model.

```
void pisces_switch::handle_credit(event *ev)
{
  pisces_credit* credit = static_cast<pisces_credit*>(ev);
  out_buffers_[credit->port()]->handle_credit(credit);
}

void pisces_switch::handle_payload(event *ev)
{
  pisces_payload* payload = static_cast<pisces_payload*>(ev);
  router_->route(payload);
  xbar_->handle_payload(payload);
}
```

The arriving event is sent to either a credit handler or a payload handler, which is configured during simulation setup. If a packet, the router object selects the next destination (port). The packet is then passed to the crossbar for arbitration.

7.8 Topology

Of critical importance for the network modeling is the topology of the interconnect. Common examples are the torus, fat tree, or butterfly. To understand what these topologies are, there are many resources on the web. Regardless of the actual structure as a torus or tree, the topology should present a common interface to the interconnect and NIC for routing messages. Here we detail the public interface.

7.8.1 Basic Topology

Not all topologies are "regular" like a torus. Ad hoc computer networks (like the internet) are ordered with IP addresses, but don't follow a regular geometric structure. The abstract topology base class is intended to cover both cases. Irregular or arbitrary topology connections are not fully supported yet.

The most important functions in the `topology` class are

```
class topology {
...
virtual bool uniform_network_ports() const = 0;

virtual bool uniform_switches_non_uniform_network_ports() const = 0;

virtual bool uniform_switches() const = 0;

virtual void connected_outports(switch_id src, std::vector<topology::connection>& conns) const = 0;

virtual void configure_individual_port_params(switch_id src,
      sprockit::sim_parameters* switch_params) const = 0;

virtual in num_switches() const = 0;

virtual int num_nodes() const = 0;

virtual int num_endpoints() const = 0;

virtual int max_num_ports() const = 0;

virtual switch_id netlink_to_injection_switch(node_id nodeaddr, uint16_t& switch_port) const = 0;

virtual switch_id netlink_to_ejection_switch(node_id nodeaddr, uint16_t& switch_port) const = 0;
```

```
25
26  virtual void configure_vc_routing(std::map<routing::algorithm_t, int>& m) const = 0;
27
28  virtual switch_id node_to_ejection_switch(node_id addr, int& port) const = 0;
29
30  virtual int minimal_distance(switch_id src, switch_id dst) const = 0;
31
32  virtual int num_hops_to_node(node_id src, node_id dst) const = 0;
33
34  virtual void nodes_connected_to_injection_switch(switch_id swid,
35                       std::vector<injection_port>& nodes) const = 0;
36
37  virtual void nodes_connected_to_ejection_switch(switch_id swid,
38                       std::vector<injection_port>& nodes) const = 0;
39
40  virtual void minimal_route_to_switch(
41   switch_id current_sw_addr,
42   switch_id dest_sw_addr,
43   routable::path& path) const = 0;
44
45  virtual bool node_to_netlink(node_id nid, node_id& net_id, int& offset) const = 0;
```

These functions are documented in the `topology.h` header file. The first few functions just give the number of switches, number of nodes, and finally which nodes are connected to a given switch. Each compute node will be connected to an injector switch and an ejector switch (often the same switch). The topology must provide a mapping between a node and its ejection and injection points. Additionally, the topology must indicate a port number or offset for the injection in case the switch has many nodes injecting to it. The most important thing to distinguish here are `node_id` and `switch_id` types. These are typedefs that distinguish between a switch in the topology and a node or network endpoint.

Besides just forming the connections, a topology is responsible for routing. Given a source switch and the final destination, a topology must fill out path information.

```
1  struct path {
2      int outport;
3      int vc;
4      int geometric_id;
5      sprockit::metadata_bits<uint32_t> metadata;
6  }
```

The most important information is the outport, telling a switch which port to route along to arrive at the destination. For congestion models with channel dependencies, the virtual channel must also be given to avoid deadlock. In general, network switches and other devices should be completely topology-agnostic. The switch is responsible for modeling congestion within itself - crossbar arbitration, credits, outport multiplexing. The switch is not able to determine for itself which outport to route along. The topology tells the switch which port it needs and the switch determines what sort of congestion delay to expect on that port. This division of labor is complicated a bit by adaptive routing, but remains essentially the same. More details are given later.

7.9 Router

The router has a simple public interface

```
1  class router {
2  ...
```

```
3     virtual void route(packet* pkt);
4
5     virtual void route_to_switch(switch_id sid, routable::path& path) = 0;
6   ...
7   };
```

Different routers exist for the different routing algorithms: minimal, valiant, ugal. The router objects are specific to a switch and can therefore store state information. However, the router should query the topology object for any path-specific information, e.g.

```
1   void minimal_router::route_to_switch(switch_id sid, routable::path& path)
2   {
3     top_->minimal_route_to_switch(my_addr_, sid, path);
4   }
```

For adaptive routing, a bit more work is done. Each router is connect to a switch object which holds all the information about queue lengths, e.g.

```
1   int test_length = get_switch()->queue_length(paths[i].outport);
```

allowing the router to select an alternate path if the congestion is too high.

Chapter 8

A Custom Object: Beginning To End

Suppose we have brilliant design for a new topology we want to test. We want to run a simple test *without* having to modify the SST/macro build system. We can create a simple external project that links the new topology object to SST/macro libraries. The Makefile can be found in `tutorials/programming/topology`. You are free to make *any* Makefile you want. After SST/macro installs, it creates compiler wrappers `sst++` and `sstcc` in the chosen `bin` folder. These are essentially analogs of the MPI compiler wrappers. This configures all include and linkage for the simulation.

We want to make an experimental topology in a ring. Rather than a simple ring with connections to nearest neighbors, though, we will have "express" connections that jump to switches far away.

We begin with the standard typedefs.

```
1  #include <sstmac/hardware/topology/structured_topology.h>
2
3  namespace sstmac {
4  namespace hw {
5
6  class xpress_ring :
7    public structured_topology
8  {
9   public:
10   typedef enum {
11     up_port = 0,
12     down_port = 1,
13     jump_up_port = 2,
14     jump_down_port = 3
15   } port_t;
16
17   typedef enum {
18     jump = 0, step = 1
19   } stride_t;
```

Packets can either go to a nearest neighbor or they can "jump" to a switch further away. Each switch in the topology will need four ports for step/jump going up/down. The header file can be found in `tutorials/programm/topology/xpressring.h`. We now walk through each of the functions in turn in the source in the topology public interface. We got some functions for free by inheriting from `structured_topology`.

We start with

```
xpress_ring::xpress_ring(sprockit::sim_parameters* params) :
  structured_topology(params)
{
  ring_size_ = params->get_int_param("xpress_ring_size");
  jump_size_ = params->get_int_param("xpress_jump_size");
}
```

determining how many switches are in the ring and how big a "jump" link is.

The topology then needs to tell objects how to connect

```
void xpress_ring::connect_objects(connectable_map& objects)
{
  for (int i=0; i < ring_size_; ++i) {
    connectable* center_obj = objects[switch_id(i)];

    switch_id up_idx((i + 1) % ring_size_);
    connectable* up_partner = find_object(objects, cloner, up_idx);
    center_obj->connect(up_port, down_port, connectable::network_link, up_partner);

    switch_id down_idx((i + ring_size_ - 1) % ring_size_);
    connectable* down_partner = find_object(objects, cloner, down_idx);
    center_obj->connect_mod_at_port(down_port, up_port, connectable::network_link,
                                    down_partner);

    switch_id jump_up_idx((i + jump_size_) % ring_size_);
    connectable* jump_up_partner = find_object(objects, cloner, jump_up_idx);
    center_obj->connect(jump_up_port, jump_down_port, connectable::network_link,
                                    jump_up_partner);

    switch_id jump_down_idx((i + ring_size_ - jump_size_) % ring_size_);
    connectable* jump_down_partner = find_object(objects, cloner,
                                         jump_down_idx);
    center_obj->connect(jump_down_port, jump_up_port, connectable::network_link,
                                    jump_down_partner);
  }
}
```

We loop through every switch in the ring and form +/− connections to neighbors and +/− connections to jump partners. Each of the four connections get a different unique port number. We must identify both the outport port for the sender and the input port for the receiver.

To compute the distance between two switches

```
int xpress_ring::num_hops(int total_distance) const
{
  int num_jumps = total_distance / jump_size_;
  int num_steps = total_distance % jump_size_;
  int half_jump = jump_size_ / 2;
  if (num_steps > half_jump) {
    //take an extra jump
    ++num_jumps;
    num_steps = jump_size_ - num_steps;
  }
  return num_jumps + num_steps;
}

int
xpress_ring::minimal_distance(
  const coordinates& src_coords,
  const coordinates& dest_coords) const
{
  int src_pos = src_coords[0];
  int dest_pos = dest_coords[0];
  int up_distance = abs(dest_pos - src_pos);
```

```
  int down_distance = abs(src_pos + ring_size_ - dest_pos);

  int total_distance = std::max(up_distance, down_distance);
  return num_hops(total_distance);
}
```

Essentially you compute the number of jumps to get close to the final destination and then the number of remaining single steps.

For computing coordinates, the topology has dimension one.

```
switch_id xpress_ring::get_switch_id(const coordinates& coords) const
{
  return switch_id(coords[0]);
}

void xpress_ring::get_productive_path(
  int dim,
  const coordinates& src,
  const coordinates& dst,
  routable::path& path) const
{
  minimal_route_to_coords(src, dst, path);
}

void xpress_ring::compute_switch_coords(switch_id swid, coordinates& coords) const
{
  coords[0] = int(swid);
}
```

Thus the coordinate vector is a single element with the `switch_id`.

The most complicated function is the routing function.

```
void xpress_ring::minimal_route_to_coords(
  const coordinates& src_coords,
  const coordinates& dest_coords,
  routable::path& path) const
{
  int src_pos = src_coords[0];
  int dest_pos = dest_coords[0];

  //can route up or down
  int up_distance = abs(dest_pos - src_pos);
  int down_distance = abs(src_pos + ring_size_ - dest_pos);
  int xpress_cutoff = jump_size_ / 2;
```

First we compute the distance in the up and down directions. We also compute the cutoff distance where it is better to jump or step to the next switch. If going up is a shorter distance, we have

```
  if (up_distance <= down_distance) {
    if (up_distance > xpress_cutoff) {
      path.outport = jump_up_port;
      path.dim = UP;
      path.dir = jump;
      path.vc = 0;
    }
    else {
      path.outport = up_port;
      path.dim = UP;
      path.dir = step;
      path.vc = 0;
    }
  }
```

We then decide if it is better to step or jump. We do not concern ourselves with virtual channels here and just set it to zero. Similarly, if the down direction in the ring is better

```
  else {
    if (down_distance > xpress_cutoff) {
      path.outport = jump_down_port;
      path.dim = DOWN;
      path.dir = jump;
      path.vc = 0;
    }
    else {
      path.outport = down_port;
      path.dim = DOWN;
      path.dir = step;
      path.vc = 0;
    }
  }
}
```

For adaptive routing, we need to compute productive paths. In this case, we only have a single dimension so there is little adaptive routing flexibility. The only productive paths are the minimal paths.

```
void xpress_ring::get_productive_path(
  int dim,
  const coordinates& src,
  const coordinates& dst,
  routable::path& path) const
{
  minimal_route_to_coords(src, dst, path);
}
```

We are now ready to use our topology in an application. In this case, we just demo with the built-in MPI ping all program from SST/macro. Here every node in the network sends a point-to-point message to every other node. There is a parameter file in the `tutorials/programming/toplogy` folder. To specify the new topology

```
# Topology
topology.name = xpress
topology.xpress_ring_size = 10
topology.xpress_jump_size = 5
```

with application launch parameters

```
# Launch parameters
launch_indexing = block
launch_allocation = first_available
launch_cmd_app1 = aprun -n10 -N1
launch_app1 = mpi_test_all
```

The file also includes a basic machine model.

After compiling in the folder, we produce an executable `runsstmac`. Running the executable, we get the following

```
Estimated total runtime of           0.00029890 seconds
SST/macro ran for        0.4224 seconds
```

where the SST/macro wall clock time will vary depending on platform. We estimate the communication pattern executes and finishes in 0.30 ms. Suppose we change the jump size to a larger number. Communication between distant nodes will be faster, but communication between medium-distance nodes will be slower. We now set `jump_size = 10` and get the output

```
Estimated total runtime of           0.00023990 seconds
SST/macro ran for       0.4203 seconds
```

We estimate the communication pattern executes and finishes in 0.24 ms, a bit faster. Thus, this communication pattern favors longer jump links.

Chapter 9

How SST/macro Launches

It is useful for an intuitive understanding of the code to walk through the steps starting from `main` and proceeding to the discrete event simulation actually launching. The code follows these basic steps:

- Configuration of the simulation via environment variables, command line parameters, and the input file
- Building and configuration of simulator components
- Running of the actual simulation

We can walk through each of these steps in more detail.

9.1 Configuration of Simulation

The configuration proceeds through the following basic steps:

- Basic initialization of the `parallel_runtime` object from environment variables and command line parameters
- Processing and parallel broadcast of the input file parameters
- Creation of the simulation `manager` object
- Detailed configuration of the `manager` and `parallel_runtime` object

The first step in most programs is to initialize the parallel communication environment via calls to MPI_Init or similar. Only rank 0 should read in the input file to minimize filesystem traffic in a parallel job. Rank 0 then broadcasts the parameters to all other ranks. We are thus left with the problem of wanting to tune initialization of the parallel environment via the input file, but the input file is not yet available. Thus, we have an initial bootstrap step where the all parameters affecting initialization of the parallel runtime must

be given either via command line parameters or environment variables. These automatically get distributed to all processes via the job launcher. Most critically the environment variable `SSTMC_PARALLEL` takes on values of `serial` or `mpi`.

As stated above, only rank 0 ever touches the filesystem. A utility is provided within the Sprockit library for automatically distributing files via the `parallel_build_params` function within `sim_parameters`. Once broadcast, all ranks now have all they need to configure, setup, and run. Some additional processing is done here to map parameters. If parameters are missing, SST/macro may fill in sensible defaults at this stage. For deprecated parameters, SST/macro also does some remapping to ensure backwards compatibility.

After creation of the `manager` object, since all of the parameters even from the input file are now available, a more detailed configuration of the `manager` and `parallel_runtime` can be done.

9.2 Building and configuration of simulator components

Inside the constructor for `manager`, the simulation manager now proceeds to build all the necessary components. There are three important components to build.

- The event manager that drives the discrete event simulation
- The interconnect object that directs the creation of all the hardware components
- The generation of application objects that will drive the software events. This is built indirectly through node objects that are built by the interconnect.

9.2.1 Event Manager

The `event_manager` object is a polymorphic type that depends on 1) what sort of parallelism is being used and 2) what sort of data structure is being used. Some allowed values include `event_map` or `event_calendar` via the `event_manager` variable in the input file. For parallel simulation, only the `event_map` data structure is currently supported. For MPI parallel simulations, the `event_manager` parameter should be set to `clock_cycle_parallel`. For multithreaded simulations (single process or coupled with MPI), this should be set to `multithread`. In most cases, SST/macro chooses a sensible default based on the configuration and installation.

As of right now, the event manager is also responsible for partitioning the simulation. This may be refactored in future versions. This creates something of a circular dependency between the `event_manager` and the `interconnect` objects. When scheduling events and sending events remotely, it is highly convenient to have the partition information accessible by the event manager. For now, the event manager reads the topology information from the input file. It then determines the total number of hardware components and does the partitioning. This partitioning object is passed on to the interconnect.

9.2.2 Interconnect

The interconnect is the workhorse for building all hardware components. After receiving the partition information from the `event_manager`, the interconnect creates all the nodes, switches, and NICs the current MPI rank is responsible for. In parallel runs, each MPI rank only gets assigned a unique, disjoint subset of the components. The interconnect then also creates all the connections between components that are linked based on the topology input (see Section 7.2). For components that are not owned by the current MPI rank, the interconnect inserts a dummy handler that informs the `event_manager` that the message needs to be re-routed to another MPI rank.

9.2.3 Applications

All events generated in the simulation ultimately originate from application objects. All hardware events start from real application code. The interconnect builds a set of node objects corresponding to compute nodes in the system. In the constructor for `node` we have:

```
job_launcher_ = job_launcher::static_job_launcher(params, mgr;
```

This job launcher roughly corresponds to SLURM, PBS, or MOAB - some process manager that will allocate nodes to a job request and spawn processes on the nodes. For implementation reasons, each node grabs a reference to a static job launcher. After construction, each node will have its `init` function invoked.

```
void node::init(unsigned int phase)
{
  if (phase == 0){
    build_launchers(params_);
  }
}
```

The `build_launchers` will detect all the launch requests from the input file. After the init phases are completed, a final setup function is invoked on the node.

```
void node::schedule_launches()
{
  for (app_launch* appman : launchers_){
    schedule(appman->time(), new_callback(this, &node::job_launch, appman));
  }
}
```

The function `appman->time()` returns the time that the application launch is *requested*, not when the application necessarily launches. This corresponds to when a user would type, e.g. `srun` or `qsub` to put the job in a queue. When the time for a job launch request is reached, the callback function is invoked.

```
void node::job_launch(app_launch* appman)
{
  job_launcher_->handle_new_launch_request(appman, this);
}
```

For the default job launcher (in most cases SST/macro only simulates a single job in which case no scheduler is needed) the job launches immediately. The code for the default job launcher is:

```
ordered_node_set allocation;
appman->request_allocation(available_, allocation);
for (const node_id& nid : allocation){
  if (available_.find(nid) == available_.end()){
    spkt_throw_printf(sprockit::value_error,
                      "allocation requested node %d, but it's not available",
                      int(nid));
  }
  available_.erase(nid);
}
appman->index_allocation(allocation);

for (int& rank : appman->rank_assignment(nd->addr()){
  sw::launch_event* lev = new launch_event(appman->app_template(), appman->aid(),
                                           rank, appman->core_affinities());
  nd->handle(lev);
}
```

Here the application manager first allocates the correct number of nodes and indexes (assigns task numbers to nodes). This is detailed in the user's manual. The application manager has a launch info object that contains all the information needed to launch a new instance of the application on each node. The application manager then loops through all processes it is supposed to launch, queries for the correct node assignment, and fetches the physical node that will launch the application.

Every application gets assigned a `software_id`, which is a struct containing a `task_id` and `app_id`. The task ID identifies the process number (essentially MPI rank). The application ID identifies which currently running application instance is being used. This is only relevant where two distinct applications are running. In most cases, only a single application is being used, in which case the application ID is always one.

9.3 Running

Now that all hardware components have been created and all application objects have been assigned to physical nodes, the `event_manager` created above is started. It begins looping through all events in the queue ordered by timestamp and runs them. As stated above, all events originate from application code. Thus, the first events to run are always the application launch events generated from the launch messages sent to the nodes generated the job launcher.

Chapter 10

Statistics Collection

Statistics collection for tracking things like congestion or number of bytes sent is difficult to standardize. Stats collection must be specifically configured to different components (e.g. NIC, CPU, memory) and types of statistic (histogram, spyplot, timeline). The stats framework is therefore intended to be highly customizable based on the individual analysis being performed without too many constraints. There are a few universal features all stats objects must comply with. First, any object that collects stats must inherit from `stat_collector` contained in the header `sstmac/common/stats/stat_collector.h`. This defines a virtual interface that every stats object must comply with. Second, stats objects should not operate on any global or static data unless absolutely necessary for space constraints. This means if you have 100K nodes, e.g., each node should maintains its own histogram of message sizes. While some storage could be saved by aggregating results into a single object, in many cases the storage overhead is minimal. This is particularly important for thread safety that stats collection be done on independent, non-interfering objects. At the very end, the `stat_collector` interface defines hooks for aggregating results if you want, e.g., a global histogram for all nodes.

10.1 Setting Up Objects

We use the example here of a the network interface histogram declared in `nic.h`.

```
1 class nic
2 {
3   ...
4   stat_histogram* hist_msg_size_;
5   ...
6   nic() : hist_msg_size_(nullptr)
7   ...
```

Here the stats object is initialized to zero. The `stat_collector` object is a factory type. Thus individual stat collectors can be associated with string identifiers. For histogram, we declare in `stat_histogram.cc`

```
1 SpktRegister("histogram", stat_collector, stat_histogram);
```

Inside the constructor for `nic`, we check if the histogram stats should be activated. Although this can be done manually, a special template function is provided for simplicity.

```
hist_msg_size_ = optional_stats<stat_histogram>(parent,
        params, "message_size_histogram", "histogram");
```

This returns a nullptr if the params does not have a namespace "message_size_histogram." Otherwise it builds a stats object corresponding to the registered factory type "histogram." By returning a nullptr, the SST component can check if the stats are active. If stats are required, the same function prototype can be used with `required_stats` which then aborts if the correct parameters are not found.

The histogram constructor initializes a few parameters internally.

```
bin_size_ = params->get_quantity("bin_size");
is_log_ = params->get_optional_bool_param("logarithmic", false);
```

defining how large histogram bins are, whether the scale is logarithmic, and finally defining a file root for dumping results later. Internally in the event manager, all objects with the same file root are grouped together. Thus the `fileroot` parameter is critical for defining unique groups of stats object. This is important during simulation post-processing when the event manager wants to aggregate results from each individual node.

10.2 Dumping Data

The first set of virtual functions that every stats object must provide are

```
virtual void simulation_finished(timestamp end) = 0;

virtual void dump_local_data() = 0;

virtual void dump_global_data() = 0;
```

`simulation_finished` tells the stats object what the final time of the simulation is and allows any final post-processing to be done. This is particularly useful in time-dependent analyses. In other cases like message size histograms, it is a no-op. After the stats object has been notified of the simulation finishing, at some point the event manager will instruct it that it is safe to dump its data. The next method, `dump_local_data`, dumps the data specific to a given node. A unique filename based on the ID provided above in the `clone_me` function is created to hold the output. The last method, `dump_global_data`, dumps aggregate data for all nodes. Here a unique filename based on the file root parameter is generated. For the default histogram, a data file and gnuplot script are created.

10.3 Reduction and Aggregation

Before the `dump_global_data` function can be called, an aggregation of results must be performed. Each stats object is therefore required to provide the functions

```
virtual void reduce(stat_collector* coll) = 0;

virtual void global_reduce(parallel_runtime* rt) = 0;
```

The first function does a local reduce. The object calling the `reduce` function aggregates data into itself from input parameter `coll` The event manager automatically loops all objects registered to the same file root and reduces them into a global aggregator. Once the aggregation is complete across all local copies, a parallel global aggregation must be performed across MPI ranks. This can be the most complicated part. For histograms, this is quite easy. A histogram is just a vector of integers. The SST/macro parallel runtime object provides a set of reduce functions for automatically summing a vector. For more complicated cases, packing/unpacking of data might need to be performed or more complicated parallel operations. Once the global reduce is done, the event manager is now safe to call `dump_global_data`. When developing new stats we recommend running medium-sized jobs as a single thread, multi-threaded, and in MPI parallel to confirm the answer is the same.

For the histogram, the reduce functions are quite simple

```
void stat_histogram::reduce(stat_collector *coll)
{
  stat_histogram* other = safe_cast(stat_histogram, coll);

  /** make sure we have enough bins to hold results */
  int max_num = std::max(counts_.size(), other->counts_.size());
  if (max_num > counts_.size()){
    counts_.resize(max_num);
  }

  /** loop all bins to aggregate results */
  int num_bins = other->counts_.size();
  for (int i=0; i < num_bins; ++i){
    counts_[i] += other->counts_[i];
  }
}
```

and for the global reduce

```
void stat_histogram::global_reduce(parallel_runtime* rt)
{
  int root = 0;
  /** Align everyone to have the same number of bins */
  int my_num_bins = counts_.size();
  int num_bins = rt->global_max(my_num_bins);
  counts_.resize(num_bins);

  /** Now global sum the data vector */
  rt->global_sum(&counts_[0], num_bins, root);
}
```

10.4 Storage Contraints

In some cases, storage constraints prevent each node from having its own copy of the data. This is particularly important for the fixed-time quanta charts which generate several MB of data even in the reduced, aggregated form. In this case it is acceptable to operate on global or static data. However, as much as possible, you should maintain the *illusion* of each component having an individual copy. For example, a NIC should not declare

```
class nic {
 ...
 static ftq_calendar* ftq_;
```

but instead

```
class nic {
 ...
 ftq_calendar* ftq_;
```

Inside the `ftq_calendar` object you can then declare

```
class ftq_calendar {
 ...
 static thread_lock lock_;
 static std::vector<ftq_epoch> results_;
```

which creates a static, aggregated set of results. The `ftq_calendar` must ensure thread-safety itself via a thread-lock.