

SST/macro 6.0: Developer's Reference

Sandia National Labs
Livermore, CA

August 10, 2016



Contents

| | | |
|----------|-------------------------------------------------|-----------|
| 1 | Introduction | 4 |
| 1.1 | Use of C++/Boost | 4 |
| 1.2 | Polymorphism and Modularity | 4 |
| 1.3 | Most Important Style and Coding Rules | 5 |
| 1.4 | Code Style Reformatting | 6 |
| 1.5 | Memory Allocation | 6 |
| 2 | SProCKit | 7 |
| 2.1 | Debug | 7 |
| 2.2 | Serialization | 8 |
| 2.3 | Keyword Registration | 10 |
| 3 | SST/macro Classes | 11 |
| 3.1 | Class Style | 11 |
| 3.1.1 | Basic Class | 11 |
| 3.1.2 | Inheritance and Parent Classes | 12 |
| 3.1.3 | Exceptions | 14 |
| 3.2 | Factory Types | 14 |
| 3.2.1 | Usage | 15 |
| 3.2.2 | Base Class | 16 |
| 3.2.3 | Child Class | 17 |
| 3.2.4 | External Linkage | 18 |

| | | |
|----------|--------------------------------------------------------------|-----------|
| 4 | Discrete Event Simulation | 19 |
| 4.1 | Event Managers | 19 |
| 4.1.1 | Event Handlers | 20 |
| 4.1.2 | Arbitrary Events | 21 |
| 4.1.3 | Event Heap/Map | 21 |
| 4.2 | Event Schedulers | 21 |
| 5 | Software Models | 23 |
| 5.1 | Applications and User-space Threads | 23 |
| 5.1.1 | Thread-specific storage | 24 |
| 5.2 | Libraries | 26 |
| 5.2.1 | API | 27 |
| 5.2.2 | Service | 28 |
| 6 | Hardware Models | 29 |
| 6.1 | Connectables | 29 |
| 6.2 | Interconnect | 30 |
| 6.3 | Node | 31 |
| 6.4 | Network Interface (NIC) | 32 |
| 6.5 | Memory Model | 33 |
| 6.6 | Network Switch | 33 |
| 6.7 | Topology | 33 |
| 6.7.1 | Basic Topology | 33 |
| 6.7.2 | Structured Topology | 35 |
| 6.8 | Router | 35 |
| 7 | A Custom Object: Beginning To End | 37 |
| 8 | How SST/macro Launches | 42 |
| 8.1 | Configuration of Simulation | 42 |
| 8.2 | Building and configuration of simulator components | 43 |
| 8.2.1 | Event Manager | 43 |
| 8.2.2 | Interconnect | 43 |
| 8.2.3 | Applications | 44 |
| 8.3 | Running | 45 |

| | | |
|----------|-------------------------------------|-----------|
| 9 | Statistics Collection | 46 |
| 9.1 | Setting Up Objects | 46 |
| 9.2 | Dumping Data | 47 |
| 9.3 | Reduction and Aggregation | 48 |
| 9.4 | Storage Constraints | 49 |

Chapter 1

Introduction

1.1 Use of C++/Boost

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.

1.2 Polymorphism and Modularity

The simulation progresses with different modules (classes) exchanging 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 messages does so via the generic function

```
1 void
2 handle(event* ev){
3   ...
4 }
```

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) carrying a data payload. 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 a 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.

1.3 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.

1.4 Code Style Reformatting

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.

1.5 Memory Allocation

To improve performance, custom memory allocation strategies can be employed in C++. At present, a global custom `operator new` can be optionally activated which is optimized for large pages and memory pools. At present, no class-specific implementation of `operator new` is used. However, we may soon implement custom allocation techniques to improve things like cache/TLB efficiency. This is the only major change expected to SST/macro that would affect externally developed modules - and even here the expected modifications would be quite small. Because custom allocation schemes may be used, all externally developed code should use `operator new`, rather than *not* `malloc` or `mmap`, unless allocating very large arrays.

Chapter 2

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.

2.1 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.

```
1 launch.h:
2 DeclareDebugSlot(job_launch);
3
4 launch.cc
5 RegisterDebugSlot(job_launch);
```

2.2 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

```
1 struct point {
2     int x;
3     int y;
4 }
5 point pt;
6 pt.x = 0;
7 pt.y = 2;
8 int niter = 5;
9 std::string str = "hello";
```

We can serialize them to a buffer

```
1 spkt_serializer ser;
2 ser.set_mode(spkt_serializer::PACK);
3 ser.init(new char[512]);
4 ser & pt;
5 ser & niter;
6 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

```
1 char* buf = new char[512];
2 MPI_Recv(buf, ...)
3 spkt_serializer;
4 ser.set_mode(spkt_serializer::UNPACK);
5 ser.init(buf);
6 ser & pt;
7 ser & niter;
8 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.

```

1 spkt_serializer ser;
2 ser.reset();
3 ser.set_mode(spkt_serializer::SIZER);
4 ser & pt;
5 ser & niter;
6 ser & str;
7 int size = ser.sizer.size(); //would be 17 for example above
8 char* buf = new char[size];
9 ...

```

The known size can now be used safely in serialization.

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

```

1 namespace my_ns {
2 class my_object :
3     public serializable,
4     public serializable_type<my_object>
5 {
6     ImplementSerializable(my_object)
7     ...
8     void
9     serialize_order(spkt_serializer& ser);
10    ...
11 };
12 }

```

The serialization interface requires two inheritances. The first inheritance from `serializable` is the one “visible” to the serializer. This inheritance forces the object to define a `serialize_order` function. The second inheritance is used in registering a type descriptor. 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. There are cleaner ways of doing this in terms of the interface, but the current setup is chosen for safety. The forced inheritance allows more safety checks to ensure types are being set up and used correctly.

In the source file, the type descriptor must be registered. This is done through the macro

```

1 DeclareSerializable(my_ns::my_object);

```

This macro should exist in the global namespace. All that remains now is defining the `serialize_order` in the source file

```

1 void
2 my_object::serialize_order(spkt_serializer& ser)
3 {
4     ser & my_int_;
5     set << my_double_;
6     ...
7 }

```

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

```

1 class parent_object :
2     public serializable
3 {
4     ...
5     void

```

```

6   serialize_order(spkt_serializer& ser);
7   ...
8   };
9
10  class my_object :
11      public parent_object,
12      public serializable_type<my_object>
13  {
14      ImplementSerializable(my_object)
15      ...
16      void
17          serialize_order(spkt_serializer& ser);
18      ...
19  };

```

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. 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

```

1  void
2  my_object::serializer_order(spkt_serializer& ser)
3  {
4      parent_object::serialize_order(ser);
5      ...
6  }

```

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

2.3 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 requires all allowed values for input parameters to be declared. 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:

```

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

```

This registers some basic keywords that might be used in a 3D grid application.

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.

```

1  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 3

SST/macro Classes

3.1 Class Style

3.1.1 Basic Class

Most classes are manually managed, being explicitly deleted. Whenever possible, smart pointers should be avoided since they create thread-safety headaches. Classes in SST/macro can inherit from the top-level class `ptr_type`, which exists in namespace `sprockit`. The examples here can be found in the code repository in `tutorials/programming/basic`.

To begin, you just need to include the appropriate header file, declare the namespace desired, and start the class declaration (see `illustration.h`).

```
1 #include <sprockit/ptr_type.h>
2
3 namespace sstmac {
4 namespace tutorial {
5
6 class illustration :
7     public sprockit::ptr_type
8 {
```

To simplify reading of the code (i.e. not typing `sprockit::refcount_ptr` everywhere), every class is required to typedef itself.

```
1     public:
2     typedef sprockit::refcount_ptr<illustration> ptr;
3     typedef sprockit::refcount_ptr<const illustration> const_ptr;
```

The reference-counted pointer type can now be used as `illustration::ptr`.

After declaring public typedefs, the public function interface can be declared. Every `ptr_type` class that is instantiated must implement a `to_string` function, mostly used for debugging purposes.

```
1     public:
2     std::string
3     to_string() const {
4         return "message class";
5     }
```

Public set/get functions can be added for member variables, if desired. We have generally followed `snake_case`, using lower-case letters and underscores.

```
1  std::string
2  message() const {
3      return message_;
4  }
5
6  void
7  set_message(const std::string& msg){
8      message_ = msg;
9  }
```

Unfortunately, amongst the various developers, a uniform style was never agreed. Some accessor functions could be called `get_message`, but most avoid the prefix and would be called just `message`. After any `private/protected` functions, member variables can be declared. SST/macro developers have not historically been very strict with use of `protected` vs. `private` for members, typically using `protected` as a default for convenience.

NOTE: We already have our first style violation here. The entire class is implemented here in a header file. In general, unless a trivial function like set/get, put the function implementation into a `.cc` file.

3.1.2 Inheritance and Parent Classes

We can also work through an example with inheritance. If you are going to use smart pointers, inheritance is somewhat tricky. Here we demonstrate in detail how to use smart pointers with virtual inheritance, which is a scenario that often occurs with the `sst_message` class. Many classes will never actually be instantiated in SST/macro- they are virtual classes defining an abstract interface. In the file `gem.h`, we define the `gem` abstract interface. Again, we include the header, declare namespaces, and declare the class.

```
1  #include <sprockit/ptr_type.h>
2
3  namespace sstmac {
4      namespace tutorial {
5
6      class gem :
7          virtual public sprockit::ptr_type
8      {
```

We note here the use of virtual inheritance, which will become important later. In general, you should almost always use virtual inheritance from `ptr_type`. Even though this class is not a complete type (pure virtual functions), we still need the public pointer typedefs.

```
1  public:
2      typedef sprockit::refcount_ptr<gem> ptr;
3      typedef sprockit::refcount_ptr<const gem> const_ptr;
```

We then define the abstract `gem` interface

```
1  public:
2      virtual int
3      value() const = 0;
4
5      virtual ~gem(){}
```

There is no static `construct` function! The static `construct` function is a signal that the class is a *complete-type*, with all virtual methods implemented. Because of C++ destructors for abstract classes, you *must* define a virtual destructor for every parent class - even if it does nothing. Without this, the memory management system will not work correctly. Many tutorials on the subtleties of virtual destructors can be found online.

We can repeat the same process for a new abstract interface called `mineral`. Declare the class:

```
1 #include <sprockit/ptr_type.h>
2
3 namespace sstmac {
4     namespace tutorial {
5
6     class mineral :
7         virtual public sprockit::ptr_type
8     {
```

Again, we use virtual inheritance. We declare public typedefs:

```
1     public:
2     typedef sprockit::refcount_ptr<mineral> ptr;
3     typedef sprockit::refcount_ptr<const mineral> const_ptr;
```

We then define the abstract `mineral` interface:

```
1     public:
2     virtual std::string
3     structure() const = 0;
4
5     virtual ~mineral(){}
```

Now we want to create a specific instance of a `gem` and `mineral`.

```
1 #include "gem.h"
2 #include "mineral.h"
3
4 namespace sstmac {
5     namespace tutorial {
6
7     class diamond :
8         public gem,
9         public mineral
10    {
```

The reason for using virtual inheritance is now evident. The `diamond` class is going to implement both the `gem` and `mineral` interfaces. However, both classes separately inherit from `ptr_type`. We thus have the multiple inheritance diamond problem. For this reason, both parent classes must virtually inherit from `ptr_type`. Most of this discussion is just a basic C++ tutorial without really being SST/macro specific. However, this diamond problem is so ubiquitous with `ptr_type`, it merits a discussion here in the programmer's reference.

We can now fill out the class. First, we have the public typedefs

```
1     public:
2     typedef sprockit::refcount_ptr<diamond> ptr;
3     typedef sprockit::refcount_ptr<const diamond> const_ptr;
```

The `ptr_type` class requires we add a `to_string` function

```
1     std::string
2     to_string() const {
3         return "diamond";
4     }
```

We now complete the gem interface

```
1 int
2 value() const {
3     return num_carats_ * 100;
4 }
```

and the mineral interface

```
1 std::string
2 structure() const {
3     return "tetrahedral carbon";
4 }
```

And the constructor

```
1     diamond(num_carats)
2     : num_carats_(num_carats)
3     {
4     }
```

Finally, we declare member variables

```
1 protected:
2     int num_carats_;
```

3.1.3 Exceptions

The `ptr_type.h` header file automatically includes a common set of exceptions, all in namespace `sprockit`. The most notable are:

- `value_error`: A parameter value is wrong
- `unimplemented_error`: A place-holder to indicate this function is not valid or not yet done
- `illformed_error`: A catch-all for anything that fails a sanity check
- `null_error`: A null object was received

SProCKit provides a macro for throwing exceptions. It should always be used since they provide extra metadata to the exception like file and line number. The macro, `spkt_throw_printf`, takes two mandatory arguments: the exception type and an exception message. An arbitrary number of arguments can be given, that get passed a `printf` invocation, e.g.

```
1 spkt_throw_printf(value_error, "invalid number of carats %d", num_carats_);
```

3.2 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`.

3.2.1 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
10    main(int argc, char **argv)
11    {
```

The details of declaring and using external apps is found in the user's manual. From here it should be apparent that we defined a new application with name `rob_reiner` which is invoked via the `main` function. 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:

```
1 appl.name = rob_reiner
2 biggest_fan = jeremy_wilke
3 actor_name = patinkin
4 sword_hand = right
```

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

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

If we change the parameters:

```
1 appl.name = rob_reiner
2 biggest_fan = jeremy_wilke
3 actor_name = guest
4 num_fingers = 6
```

we now get the output

```
1 You've been chasing me your entire life only to fail now.
2 I think that's the worst thing I've ever heard. How marvelous.
3 Estimated total runtime of          0.00000000 seconds
4 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.

3.2.2 Base Class

To declare a new factory type, you must include the factory header file and inherit from the base class `factory_type`.

```
1 #include <sprockit/factories/factory.h>
2
3 namespace sstmac {
4     namespace tutorial {
5
6     class actor :
7     public sprockit::factory_type
8     {
```

We now define the public interface for the actor class

```
1     public:
2     virtual void
3     act() = 0;
4
5     virtual ~actor(){}
6 }
```

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 implement

```
1     virtual void
2     init_factory_params(sprockit::sim_parameters* params);
3 }
```

In general, constructors for factory types do no work. Reading parameters and initializing values happens entirely in this function.

The parent class has a single member variable

```
1     protected:
2     std::string biggest_fan_;
```

After finishing the class, we need to invoke a macro

```
1 DeclareFactory(actor);
```

making SST/macro aware of the new factory type.

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

```
1 namespace sstmac {
2     namespace tutorial {
3
4     void
5     actor::init_factory_params(sprockit::sim_parameters* params)
6     {
7         biggest_fan_ = params->get_param("biggest_fan");
8     }
9 }
```

We initialize the member variable from the parameter object. For cloning, we have instead

```

1 void
2 actor::clone_into(actor* cln) const
3 {
4     cln->biggest_fan_ = biggest_fan_;
5 }

```

which initializes from the existing object being cloned.

We additionally need a macro

```

1 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.

3.2.3 Child Class

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

```

1 #include "actor.h"
2
3 namespace sstmac {
4     namespace tutorial {
5
6     class mandy_patinkin :
7     public actor
8     {

```

We have a single member variable

```

1     protected:
2         std::string sword_hand_;

```

This is a complete type that can be instantiated.

To create the class we will need the functions

```

1 virtual void
2 init_factory_params(sprockit::sim_parameters* params);
3
4 virtual actor*
5 clone() const;

```

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

```

1 virtual void
2 act();

```

For doing the clone work, we need a protected helper function

```

1     protected:
2     void
3     clone_into(mandy_patinkin* cln) const;

```

Moving to the implementation, we must first register the new type using the macro

```

1 namespace sstmac {
2     namespace tutorial {
3
4     SpktRegister("patinkin", actor, mandy_patinkin,
5         "He's on one of those shows now... NCIS? CSI?");

```

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. Whatever string value is registered here will be used in the input file to create the type. We can now implement the functions

```

1 void
2 mandy_patinkin::init_factory_params(sprockit::sim_parameters* params)
3 {
4     sword_hand_ = params->get_param("sword_hand");
5
6     if (sword_hand_ == "left"){
7         spkt_throw(value_error, "I am not left handed!");
8     }
9     else if (sword_hand_ != "right"){
10         spkt_throw_printf(value_error,
11             "Invalid hand specified: %s",
12             sword_hand_.c_str());
13     }
14     actor::init_factory_params(params);
15 }

```

We are safe to throw exceptions here since we are not in the constructor. The child class must invoke the parent class method. Finally, we specify the acting behavior

```

1 void
2 mandy_patinkin::act()
3 {
4     std::cout << "Hello. My name is Inigo Montoya. You killed my father. Prepare to die!"
5         << std::endl;
6 }

```

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

3.2.4 External Linkage

If you glance at the Makefile, you will see how and why the executable is created. A compiler wrapper `sst++` points the makefile to the SST/macro libraries, including a library `libsstmac_main` that actually implements the `main` routine and the SST/macro driver. By creating another executable, arbitrary code can be linked together with the core SST/macro framework. The same SST/macro driver is invoked in the external executable as would be invoked by the default main executable.

Chapter 4

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 that cooperate in driving the simulation, which we now describe.

4.1 Event Managers

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

```
1 virtual void  
2 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  
2 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)

```

1 namespace sstmac {
2
3 class event_queue_entry
4 {
5 public:
6     virtual void
7         execute() = 0;
8
9     ...
10 };

```

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.

4.1.1 Event Handlers

In most cases, the event is represented as a message sent to an object called an `event_handler` at a specific simulation time. In handling the message, the event handlers change their internal state and may cause more events by scheduling new messages 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

```

1 void
2 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

```

1 void
2 schedule(const timestamp &t,
3         event_handler* handler,
4         event* ev);

```

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

```

1 void
2 handler_event_queue_entry::execute()
3 {
4     if (!canceled_) {
5         handler_->handle(ev_to_deliver_);
6     }
7 }

```

4.1.2 Arbitrary Events

In some cases, it can be inconvenient (and inefficient) to require every event to be funneled through an `event_handler` type. A generic macro for creating events from any class member function is provided in the file `event_callback.h`. For example, the MPI server creates an event

```
1 mpi_queue_recv_request* req = next_request();
2 event* ev = new_event(this, &mpi_queue::start_recv, req);
```

The new event macro takes as first argument an object and second argument a member function pointer to be invoked when the event is run. The remaining arguments are an arbitrary-length list of parameters of any type - they don't need to be messages. These parameters should match the member function prototype. For example, the prototype

```
1 void
2 mpi_queue::start_recv(mpi_queue_recv_request* req)
3 {
4     ...
5 }
```

takes a single `recv_request` object as input. This eases the programming burden in two ways. First, it avoids having to always create `event_handler` types. Without template events, you would have to create a new class that inherits from `event_handler` that performs a single, desired action. Second, it helps direct messages to the right place. A single `event_handler` might process many different types of events or objects. If every event went to a single `handle` method, the handle method would need either a long if-else block or switch to sort messages. The event would also need to be dynamic cast to the correct type. By creating event functors, the message can be immediately directed to the correct type and correct action.

For generic events, one must ensure the event is scheduled to the same node and does not cross any network boundaries. Generic events are not compatible with parallel simulation. The event created must be run on the same logical process.

4.1.3 Event Heap/Map

The major distinction between different event containers is the data structure 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.

4.2 Event Schedulers

The simulation is partitioned into objects that are capable of scheduling events. `event_scheduler` objects are also `event_handler` objects, although their handle methods might never be used. 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. In parallel simulation, though, the simulation must be partitioned into different scheduling units. Scheduling

units are then distributed amongst the parallel processes. The `event_scheduler` is therefore the basic unit of parallelism. Additionally, when simulating failures (in either serial or parallel), certain devices must be deactivated such as a node or network switch going down. This basically amounts to canceling all events associated with a given `event_scheduler`. The `event_scheduler` is therefore primarily a means of structuring the computation.

Chapter 5

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 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. The current solution involves thread-specific storage and the use of special C++ variables (more below).

5.1 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(...) current_mpi()->send(__VA_ARGS__)
3 ...

```

`current_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 5.1, 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 enqueueing 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.

5.1.1 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

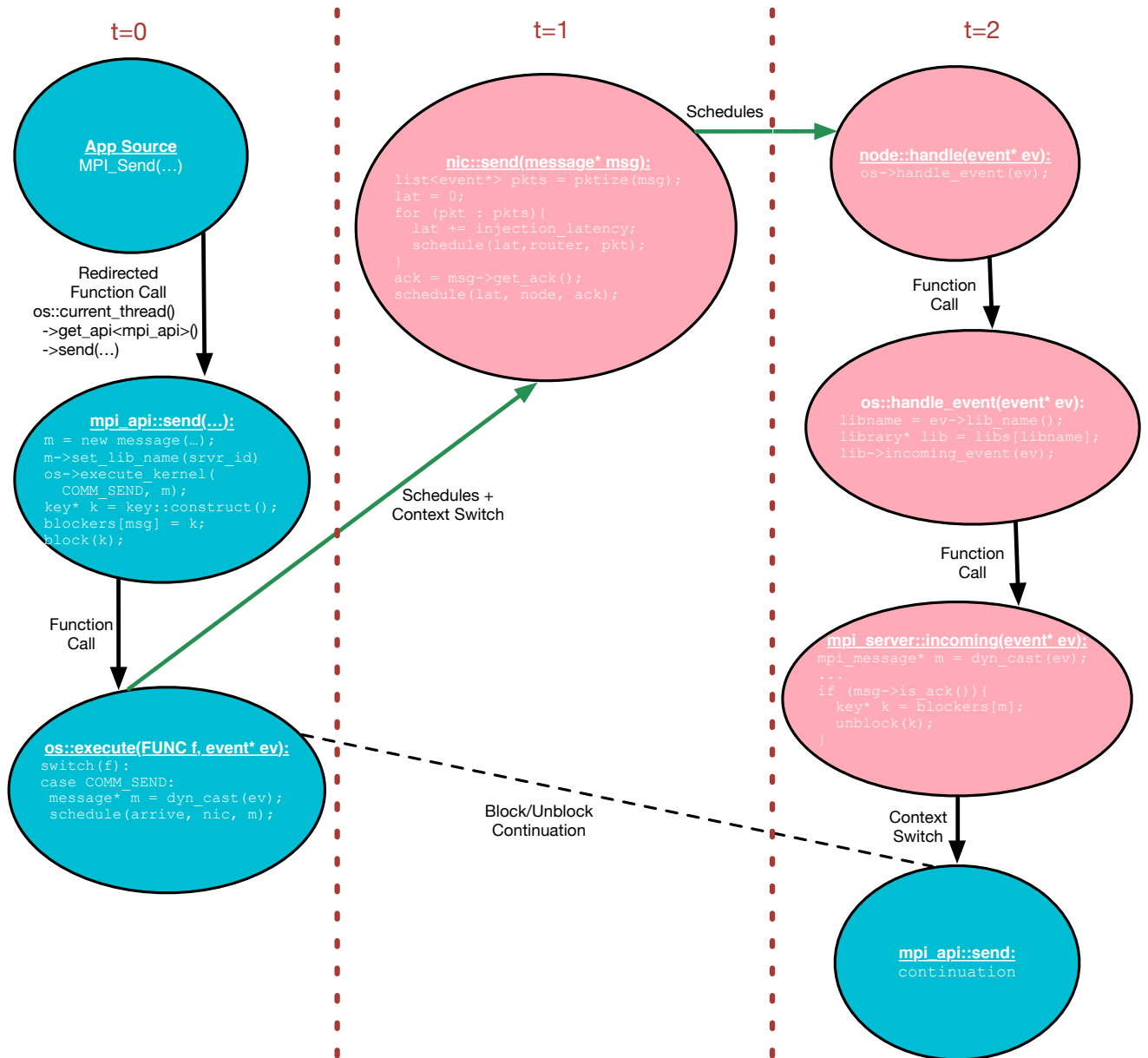


Figure 5.1: Flow of events for completing a send operation. Shows basic function calls, block/unblock context switches, and event schedules. User-space thread (application) operations are shown in blue. Main event thread (OS/kernel) operations are shown in pink.

| | 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 <code>handle_event</code> function | Function calls from OS, receives scheduled events via the <code>handle</code> function | Does not usually receive events - only blocks or unblocks | OS forwards messages to <code>incoming_event</code> 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 |

holding all state specific to that virtual thread. Rather than store MPI state in global variables, MPI state is 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.

5.2 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.

```

1  class operating_system {
2      ...
3      map<string,library*> libs_;
4      ...
5  };

```

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

```

1  event* ev = ...
2  os->execute_kernel(ty, ev);

```

or possibly

```
1 event* ev = ...
2 os->execute(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

```
1 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:

```
1 key* k = key::construct();
2 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.

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

```
1 void
2 operating_system::handle_event(event* ev) {
3     library* lib = libs_[ev->lib_name()];
4     lib->incoming_event(ev);
5 }
```

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`.

5.2.1 API

The SST/macro definition of API was alluded to in 5.1.1. 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`.

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

for which the implementation is

```
1 thread* thr = operating_system::current_thread();
2 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.

5.2.2 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.

Chapter 6

Hardware Models

6.1 Connectables

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 4, 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

```
1  class connectable
2  {
3  public:
4      virtual void
5      connect(
6          int src_outport,
7          int dst_inport,
8          connection_type_t ty,
9          connectable* mod,
10         config* cfg) = 0;
11 };
```

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:

```
1  connectable* src = ...
2  connectable* dst = ...
3  connectable::config cfg = ...
4  src->connect(inport, outport, Output, dst, &cfg);
5  dst->connect(inport, outport, Input, src, &cfg);
```

The “direction” of the link as input or output is identified by `connection_type_t`.

A certain style and set of rules is recommended for all connectables. If these rules are ignored, setting up connections can quickly 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 **topology** 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 6.7.

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 with a recursive mess. If you need a bidirectional link ($A \rightarrow B$, $B \rightarrow A$), two separate function calls should be made

```
1 A->connect(B);
2 B->connect(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.

6.2 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.

```
1 void
2 interconnect::init_factory_params(sprockit::sim_parameters* params)
3 {
4     sprockit::sim_parameters* top_params = params->get_namespace("topology");
5     topology_ = topology_factory::get_param("name", top_params);
6
7     endpoint_map nodes;
8     sprockit::sim_parameters* node_params = params->get_namespace("node");
9     sprockit::factory<connectable>* node_builder
10     = new sprockit::template_factory<connectable, node_factory>(node_params->get_param("model"));
11     topology_>build_endpoint_connectables(nodes, node_builder, partition_, rt_>me(), node_params);
12 }
```

The interconnect creates a topology and factory builder for the nodes. Note here the use of parameter namespaces to isolate parameters. This continues for the NIC and switches.

```

1  endpoint_map nics;
2  sprockit::sim_parameters* nic_params = params->get_namespace("nic");
3  sprockit::factory2<connectable>* nic_builder
4  = new sprockit::template_factory2<connectable, nic_factory>(nic_params->get_param("model"));
5  topology->build_interface_connectables(1, nics, nic_builder, partition_, rt_->me(), nic_pa

1  internal_map switches;
2  sprockit::sim_parameters* switch_params = params->get_namespace("switch");
3  sprockit::factory<connectable>* switch_builder
4  = new sprockit::template_factory<connectable, network_switch_factory>(switch_params->get_param("model"));
5  network_switch* dummy = new dist_dummy_switch(switch_id());
6  topology->build_internal_connectables(switches, switch_builder, partition_, rt_->me(), switch_params, dummy);

```

Some special parameters (dummy switches) are important for MPI parallel simulations, but these can be ignored for now.

Once these components are constructed, the topology connects them.

```

1  topology->connect_end_points(switches_, nics_);
2  topology->connect_topology(switches_);

```

6.3 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
2  execute_kernel(ami::COMP_FUNC func, event* data);
3
4  virtual void
5  execute_kernel(ami::COMM_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. The distinction between computation and hardware is subtle. Hardware operations are things like interrupts, device resets, device failures. They are not necessarily “kernels” in the standard parlance.

To illustrate a single example, we show the code that handles the function call

```

1  node->execute_kernel(ami::AMI_COMM_SEND, msg);

```

leads to

```

1  switch (func) {
2      case sstmac::sw::ami::AMI_COMM_SEND:
3      {
4          network_message* netmsg = ptr_safe_cast(network_message, data);
5          netmsg->set_fromaddr(my_id_);
6          if (netmsg->toaddr() == nodeid_) {
7              /* Intranode send */
8          }
9          else {
10             nic_->send(netmsg);
11         }
12     }

```


All dynamic casts are routed through a special macro `safe_cast` for most types or `ptr_safe_cast` for smart pointer types.

6.4 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
2  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 `null_nic`, the function is

```
1  injector_ ->handle(msg);
2  if (msg->get_needs_ack()) {
3      sst_message* acker = msg->clone_ack();
4      schedule(now(), parent_node_, acker);
5  }
```

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:

```
1  event_handler*
2  mtl_handler() const {
3      return mtl_handler_;
4  }
```

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

6.5 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.

```
1 virtual void
2 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.

6.6 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 `packet_flow` model.

```
1 packet_flow_interface* fpack = interface_cast(packet_flow_interface, ev);
2 switch (fpack->type()) {
3     case packet_flow_interface::credit: {
4         packet_flow_credit* credit = static_cast<packet_flow_credit*>(fpack);
5         out_buffers_[credit->port()->handle_credit(credit);
6         break;
7     }
8     case packet_flow_interface::payload: {
9         packet_flow_payload* payload = static_cast<packet_flow_payload*>(fpack);
10        router_->route(payload);
11        xbar_->handle_payload(payload);
12        break;
13    }
14 }
```

The arriving event is determined to either be a new packet or a credit. If a packet, the router object selects the next destination (port). The packet is then passed to the crossbar for arbitration.

6.7 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.

6.7.1 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

```

1 class topology
2 {
3
4     virtual std::vector<node_id>
5     get_nodes_connected_to_switch(switch_id swid) const = 0;
6
7     virtual long
8     num_switches() const = 0;
9
10    virtual long
11    num_nodes() const = 0;
12
13    virtual void
14    connect_objects(connectable_map& objects) = 0;
15
16    virtual switch_id
17    node_to_injection_switch(node_id nodeaddr, int& switch_port) const = 0;
18
19    virtual switch_id
20    node_to_ejection_switch(node_id nodeaddr, int& switch_port) const = 0;
21
22    virtual void
23    minimal_route_to_switch(
24        switch_id current_sw_addr,
25        switch_id dest_sw_addr,
26        routable::path& path) const = 0;
27
28    virtual int
29    num_hops_to_node(node_id src, node_id dst) 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. The most critical function for a topology is the `connect_objects` function that takes a list of objects and actually forms the links between them. 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 or internal object 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. More detail can be given with dimension and direction for some topologies (i.e. +X or -X in a torus). 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.

6.7.2 Structured Topology

The structured topology assumes a regular, ordered connecting of nodes like a torus. It is synonymous with any topology that can be mapped onto a coordinate system. This is true even of tree structures for which the coordinates define which branch/leaf. The structured topology introduces a few extra virtual functions:

```
1  class structured_topology :
2      public topology
3  {
4      ....
5      virtual int
6      ndimensions() const = 0;
7
8      virtual void
9      compute_switch_coords(switch_id swid, coordinates& coords) const = 0;
10
11     virtual void
12     minimal_route_to_coords(
13         const coordinates& src_coords,
14         const coordinates& dest_coords,
15         routable::path& path) const = 0;
16
17     virtual int
18     minimal_distance(
19         const coordinates& src_coords,
20         const coordinates& dest_coords) const = 0;
21
22     virtual void
23     get_productive_path(
24         int dim,
25         const coordinates& src,
26         const coordinates& dst,
27         routable::path& path) const = 0;
28     ...
29 };
```

The structured topology must have a well-defined number of dimensions, such as a 3D-torus or the number of levels in a tree. The topology must also be able to map any `switch_id` to a unique set of coordinates. Given two input sets of input coordinates, it should be able to perform the routing operation or compute the minimal distance between two points. Finally, for adaptive routing purposes, the topology should map routing requests to the correct path. For example, if I want to move in the X dimension in a 3D-torus, what path (port) is needed to make a productive step? It can be that any step along that dimension is unproductive - you've already arrived. The behavior is then undefined.

6.8 Router

The router has a simple public interface

```
1  class router :
2      public sprockit::factory_type
3  {
4      ...
5      virtual void
6      route(packet* pkt) = 0;
7
8      virtual void
9      minimal_route_to_node(
10         node_id node_addr,
11         routable::path& path) = 0;
```

```

12
13     virtual void
14     minimal_route_to_switch(
15         switch_id sw_addr,
16         routable::path& path) = 0;
17     ...
18 };

```

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
2  structured_router::minimal_route_to_node(
3      node_id dest_node_addr,
4      routable::path& path)
5  {
6      netlink_id endpoint_id(dest_node_addr / top->num_nodes_per_netlink());
7      //Query the topology for path info
8      switch_id ej_addr = regtop->endpoint_to_ejection_switch(endpoint_id, path.outport);
9      if (ej_addr == my_addr_) {
10         path.vc = 0;
11     }
12     else {
13         minimal_route_to_switch(ej_addr, path);
14     }
15 }
16 ...

```

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 7

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

```

1 void
2 xpress_ring::init_factory_params(sprockit::sim_parameters* params)
3 {
4     structured_topology::init_factory_params(params);
5     ring_size_ = params->get_int_param("xpress_ring_size");
6     jump_size_ = params->get_int_param("xpress_jump_size");
7 }

```

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

```

1 void
2 xpress_ring::connect_objects(connectable_map& objects)
3 {
4     for (int i=0; i < ring_size_; ++i) {
5         connectable* center_obj = objects[switch_id(i)];
6
7         switch_id up_idx((i + 1) % ring_size_);
8         connectable* up_partner = find_object(objects, cloner, up_idx);
9         center_obj->connect(up_port, down_port, connectable::network_link, up_partner);
10
11         switch_id down_idx((i + ring_size_ - 1) % ring_size_);
12         connectable* down_partner = find_object(objects, cloner, down_idx);
13         center_obj->connect_mod_at_port(down_port, up_port, connectable::network_link,
14                                         down_partner);
15
16         switch_id jump_up_idx((i + jump_size_) % ring_size_);
17         connectable* jump_up_partner = find_object(objects, cloner, jump_up_idx);
18         center_obj->connect(jump_up_port, jump_down_port, connectable::network_link,
19                             jump_up_partner);
20
21         switch_id jump_down_idx((i + ring_size_ - jump_size_) % ring_size_);
22         connectable* jump_down_partner = find_object(objects, cloner,
23                                                       jump_down_idx);
24         center_obj->connect(jump_down_port, jump_up_port, connectable::network_link,
25                             jump_down_partner);
26     }
27 }

```

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 output port for the sender and the input port for the receiver.

To compute the distance between two switches

```

1 int
2 xpress_ring::num_hops(int total_distance) const
3 {
4     int num_jumps = total_distance / jump_size_;
5     int num_steps = total_distance % jump_size_;
6     int half_jump = jump_size_ / 2;
7     if (num_steps > half_jump) {
8         //take an extra jump
9         ++num_jumps;
10        num_steps = jump_size_ - num_steps;
11    }
12    return num_jumps + num_steps;
13 }
14
15 int
16 xpress_ring::minimal_distance(
17     const coordinates& src_coords,
18     const coordinates& dest_coords) const
19 {

```

```

20     int src_pos = src_coords[0];
21     int dest_pos = dest_coords[0];
22     int up_distance = abs(dest_pos - src_pos);
23     int down_distance = abs(src_pos + ring_size_ - dest_pos);
24
25     int total_distance = std::max(up_distance, down_distance);
26     return num_hops(total_distance);
27 }

```

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.

```

1  switch_id
2  xpress_ring::get_switch_id(const coordinates& coords) const
3  {
4      return switch_id(coords[0]);
5  }
6
7  void
8  xpress_ring::get_productive_path(
9      int dim,
10     const coordinates& src,
11     const coordinates& dst,
12     routable::path& path) const
13  {
14     minimal_route_to_coords(src, dst, path);
15  }
16
17  void
18  xpress_ring::compute_switch_coords(switch_id swid, coordinates& coords) const
19  {
20     coords[0] = int(swid);
21 }

```

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

The most complicated function is the routing function.

```

1  void
2  xpress_ring::minimal_route_to_coords(
3      const coordinates& src_coords,
4      const coordinates& dest_coords,
5      routable::path& path) const
6  {
7      int src_pos = src_coords[0];
8      int dest_pos = dest_coords[0];
9
10     //can route up or down
11     int up_distance = abs(dest_pos - src_pos);
12     int down_distance = abs(src_pos + ring_size_ - dest_pos);
13     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

```

1     if (up_distance <= down_distance) {
2         if (up_distance > xpress_cutoff) {
3             path.outport = jump_up_port;
4             path.dim = UP;
5             path.dir = jump;
6             path.vc = 0;
7         }

```



```

8     else {
9         path.outport = up_port;
10        path.dim = UP;
11        path.dir = step;
12        path.vc = 0;
13    }
14 }

```

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

```

1     else {
2         if (down_distance > xpress_cutoff) {
3             path.outport = jump_down_port;
4             path.dim = DOWN;
5             path.dir = jump;
6             path.vc = 0;
7         }
8         else {
9             path.outport = down_port;
10            path.dim = DOWN;
11            path.dir = step;
12            path.vc = 0;
13        }
14    }
15 }

```

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.

```

1 void
2 xpress_ring::get_productive_path(
3     int dim,
4     const coordinates& src,
5     const coordinates& dst,
6     routable::path& path) const
7 {
8     minimal_route_to_coords(src, dst, path);
9 }

```

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/topology` folder. To specify the new topology

```

1 # Topology
2 topology.name = xpress
3 topology.xpress_ring_size = 10
4 topology.xpress_jump_size = 5

```

with application launch parameters

```

1 # Launch parameters
2 launch_indexing = block
3 launch_allocation = first_available
4 launch_appl_cmd = aprun -n10 -N1
5 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 8

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:

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

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

8.1 Configuration of Simulation

The configuration proceeds through the following basic steps:

1. Basic initialization of the `parallel_runtime` object from environment variables and command line parameters
2. Processing and parallel broadcast of the input file parameters
3. Creation of the simulation `manager` object
4. 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.

8.2 Building and configuration of simulator components

Inside the function `mgr->init_factory_params`, the simulation manager now proceeds to build all the necessary components. There are three basic classes of 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

8.2.1 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.

8.2.2 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 6.1). 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.

8.2.3 Applications

All events generated in the simulation ultimately originate from application objects. All hardware events start from real application code. To generate application objects, the manager first calls `build_apps`, which loops through and finds all the applications to be launched. It then builds a template object, which will be used to launch all the individual instances of 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 really only relevant in cases like in situ analysis where two distinct applications are running. In most cases, only a single application is being used, in which case the application ID is always zero. The simulation manager depends on an `app_manager` object that keeps track of the mapping between software IDs and the actual physical nodes that are running the apps.

To actually launch the app, we have the following code:

```

1  void
2  manager::launch_app(int appnum, timestamp start, sw::app_manager* appman)
3  {
4      appman->allocate_and_index_jobs();
5      launch_info* linfo = appman->launch_info();
6      sstmac::sw::app_id aid(appnum);
7      for (int i=0; i < appman->nproc(); ++i) {
8          node_id dst_nid = appman->node_assignment(i);
9          sstmac_runtime::register_node(aid, task_id(i), dst_nid);
10
11         hw::node* dst_node = interconnect->node_at(dst_nid);
12         if (!dst_node) {
13             // mpirun, this node belongs to someone else
14             continue;
15         }
16
17         sw::launch_message::ptr lmsg = new launch_message(linfo, sw::launch_message::ARRIVE, task_id(i));
18         int dstthread = dst_node->thread_id();
19         event_manager->ev_man_for_thread(dstthread)->schedule(start, new handler_event(lmsg, dst_node));
20     }

```

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. If a null node is returned, that indicates the physical node is owned by another MPI rank and should be skipped. Finally a launch message containing the launch info is scheduled to arrive at the node. When the node receives the launch message, it will actually create the application object and pass it off to the `operating_system` object to allocate stack resources and start it running.

8.3 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.

Chapter 9

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 maintain 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.

9.1 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_(0)
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 `init_factory_params` function for `nic`, we check if the histogram stats should be activated

```

1  if (params->has_namespace("message_sizes")){
2      params->enter_namespace("message_sizes");
3      stat_collector* hist = stat_collector_factory::get_optional_param("type", "histogram", params);
4      hist_msg_size_ = test_cast(stat_histogram, hist);
5      params->exit_namespace("message_sizes");
6  }

```

It is possible that you create your own customized histogram object. However, here we default to the histogram provided by SST/macro. The histogram constructor initializes a few parameters internally.

```

1  bin_size_ = params->get_quantity("bin_size");
2  is_log_ = params->get_optional_bool_param("logarithmic", false);
3  fileroot_ = params->get_param("fileroot");

```

defining how large histogram bins are, whether the scale is logarithmic, and finally defining a file root for dumping results later.

This only defines a template histogram object, though. Recall from earlier sections, almost everything in SST/macro builds a template object and then clones copies of it to fill out the machine.

```

1  void
2  nic::clone_into(nic* nic) const
3  {
4      if (hist_msg_size_){
5          nic->hist_msg_size_ = hist_msg_size_->clone_me(parent_->id());
6      }

```

If we have built a histogram object, create a copy of it but assign it a unique ID. This unique ID is important for the runtime to distinguish a stat collector object for node 0 from that for node 1, node 2, etc.

When the simulation begins, every component gets assigned an **event_manager**. At this time, an object must register all statistics with the runtime

```

1  void
2  nic::set_event_manager(event_manager* m)
3  {
4      if (hist_msg_size_) m->register_stat(hist_msg_size_);

```

At this point, the stats object is completely configured. The stats object copy has been assigned a unique ID and registered with the SST runtime. Each individual NIC can now start (thread-safely) collecting stats. 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.

9.2 Dumping Data

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

```

1  virtual void
2  simulation_finished(timestamp end) = 0;
3
4  virtual void
5  dump_local_data() = 0;
6
7  virtual void
8  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.

9.3 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

```
1 virtual void
2 reduce(stat_collector* coll) = 0;
3
4 virtual void
5 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

```
1 void
2 stat_histogram::reduce(stat_collector *coll)
3 {
4     stat_histogram* other = safe_cast(stat_histogram, coll);
5
6     /** make sure we have enough bins to hold results */
7     int max_num = std::max(counts_.size(), other->counts_.size());
8     if (max_num > counts_.size()){
9         counts_.resize(max_num);
10    }
11
12    /** loop all bins to aggregate results */
13    int num_bins = other->counts_.size();
14    for (int i=0; i < num_bins; ++i){
15        counts_[i] += other->counts_[i];
16    }
17 }
```

and for the global reduce

```

1 void
2 stat_histogram::global_reduce(parallel_runtime* rt)
3 {
4     int root = 0;
5     /** Align everyone to have the same number of bins */
6     int my_num_bins = counts_.size();
7     int num_bins = rt->global_max(my_num_bins);
8     counts_.resize(num_bins);
9
10    /** Now global sum the data vector */
11    rt->global_sum(&counts_[0], num_bins, root);
12 }

```

9.4 Storage Constraints

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

```

1 class nic {
2     ...
3     static ftq_calendar* ftq_;

```

but instead

```

1 class nic {
2     ...
3     ftq_calendar* ftq_;

```

Inside the `ftq_calendar` object you can then declare

```

1 class ftq_calendar {
2     ...
3     static thread_lock lock_;
4     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.