MUSIC — Multi-Simulation Coordinator

Users Manual

Örjan Ekeberg and Mikael Djurfeldt

July 23, 2010

Abstract

MUSIC is an API allowing large scale neuron simulators using MPI internally to exchange data during runtime. MUSIC provides mechanisms to transfer massive amounts of event information and continuous values from one parallel application to another. Special care has been taken to ensure that existing simulators can be adapted to MUSIC. In particular, MUSIC handles data transfer between applications that use different time steps and different data allocation strategies.

Contents

List of Figures

Chapter 1

Introduction

This document constitutes the final specification for the multi-simulation coordinator MUSIC.

1.1 Scope

MUSIC is a standard for run-time exchange of data between parallel applications in a cluster environment. The standard is designed specifically for interconnecting large scale neuronal network simulators, either with each-other or with other tools.

A typical usage example is illustrated in figure 1.1, where three applications (A, B, and C) are executing in parallel while exchanging data via MUSIC. We will refer to this as a *multi-simulation*, since the participating applications typically are neuronal simulators, or tools to support such simulators. In this example, application A produces runtime data which is then used by B and C. In addition, B and C mutually send data to each other. The data sent between applications can be either event based, such as neuronal spikes, or graded continuous values, for example membrane voltages.

The primary objective of MUSIC is to support multi-simulations where each participating application itself is a parallel simulator with the capacity to produce and/or consume massive amounts of data. This promotes *inter-operability* by allowing models written for different simulators to be simulated together in a larger system. It also enables *re-usability* of models or tools by providing a standard interface. The fact that data is spread out over a number of processors makes it non-trivial to coordinate the transfer of data so that it reaches the right destination at the right time. The task for MUSIC is to relieve the applications from handling this complexity.

1.2 Design Goals

1.2.1 Portability

The MUSIC library and utilities have been designed to run smoothly on state-of-the-art high-performance hardware. For maximal portability, the software is written in C++, which is the de facto standard for current high-end hardware.

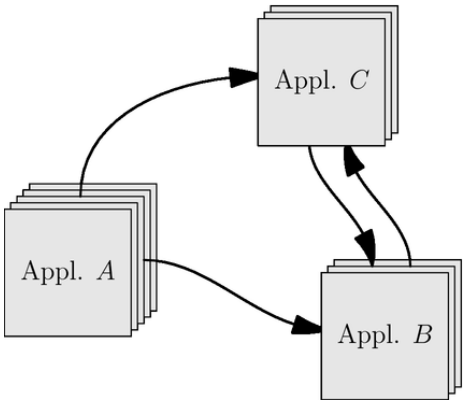

Figure 1.1: Illustration of a typical multi-simulation using MUSIC. Three applications, A, B, and C, are exchanging data during runtime.

MUSIC also provides a C-interface, making it possible for applications written in C or FORTRAN to participate in a MUSIC multi-simulation.

Most, if not all, current efforts in large scale neuronal simulations are based on the MPI standard. MUSIC is built on top of MPI, and uses it to run the different simulators. MUSIC provides means to allow each simulator to use MPI internally without interfering with the others.

MUSIC has been developed using two reference platforms: Intel-based multi-core workstations and the IBM BlueGene/L supercomputer. These platforms can be considered as two extremes, where the multi-core machine represents a small parallel environment while the BlueGene/L represents a large scale massively parallel supercomputer with special requirements. In particular, the compute nodes on the BlueGene/L do not support multiple threads or processes.

1.2.2 Simplicity

For MUSIC to be useful, it must be possible to adapt existing simulators so that they can participate in a multi-simulation without too much effort. We rely on the simulator developers to make these adaptations. An important design goal has therefore been to adapt the design to the typical structure of current simulators. It should be possible to add MUSIC library support without invasive restructuring of the existing code.

The primary requirements on an application using MUSIC is that it declares what data should be exported and imported and that it repeatedly calls a function at regular intervals during the simulation to allow MUSIC to make the actual data transfer.

1.2.3 Independence

The MUSIC interface ensures that each individual application does not need special adaptation to specific properties of other applications. The application only needs to adhere to the specification of the MUSIC interface in order to communicate with other applications performing complementary tasks. This

makes the development of MUSIC-aware software independent of what other applications it will communicate with.

We hope that this will facilitate the development of general purpose tools. For example, a researcher can develop a tool for calculating synthetic EEG from simulation data. Via MUSIC, this tool should then be useful for anybody using any neuronal simulator which supports the common MUSIC interface.

1.2.4 Performance

The MUSIC API has been designed to allow for data transport of high bandwidth and low latency within the cluster. One means of ensuring the best use of the hardware while maintaining portability is to use the facilities of MPI for communication. MPI encapsulates software optimizations for specific hardware. By basing the interface on MPI we can benefit from such optimizations.

1.2.5 Extensibility

Where possible, MUSIC allows for extensions by the application programmer. Some classes in the MUSIC API (such as the index and data maps) can be subclassed in order to provide facilities not available directly in the API.

1.3 Terminology

application We use the term *application* to denote a simulator or other program interfaced to MUSIC. Each application is a parallel program, normally running on several processors.

multi-simulation We use the term *multi-simulation* to refer to the whole parallel execution of multiple applications coordinated by MUSIC.

port Each application declares its ability to produce and consume data by publishing *ports*. Ports are named by the application and provided with information about the datatype (continuous data, spike events, messages) and mapping onto different processors. Ports are either `InputPort`s or `OutputPort`s.

connection During the setup phase, MUSIC connects pairs of ports together to form *connections*. During the runtime phase, data is transferred over the connection from the producer of the data to the consumer. While an `InputPort` can have only one connection, an `OutputPort` can be connected to multiple `InputPort`s.

data map A data map denotes the information on where data actually resides within the application. This is typically stored internally in the port data structure. Data to be transferred over a connection can be regarded as a large array distributed over multiple processors. The data map tells on what processor each data element resides and how it should be accessed.

ticks During the runtime phase, all processes in each application must make a *tick* call at regular intervals in simulated time. At these tick points, MUSIC is allowed to use MPI to transfer data between processors.

1.4 Relation to Existing Software

MUSIC is not the only software project aiming to support inter-operability between neural simulators. In this section we will briefly describe some related projects and specifically focus on how they relate to MUSIC.

PyNN PyNN is a Python package for simulator-independent specification of neuronal network models. It provides a low-level procedural API and a high-level object-oriented API. Neuronal network models which are specified using these API:s can be simulated on simulators supporting PyNN, such as Neuron and NEST.

PyNN could be extended to support multi-simulations using the MUSIC library. Such an extension would provide means for controlling the interaction between the simulator and the MUSIC library and would, for example, support publishing of named ports.

It is possible, in principle, to write Python code to directly handle communication between applications in a cluster, but such a solution would be inefficient compared to using MUSIC, and might, in the end, have to address the same problems which MUSIC provides a solution to.

Neurospaces The Neurospaces project promotes inter-operability and re-usability through the development of independent software components, some of which, together, will provide one of two alternative cores of the Genesis 3 simulator. One of the components, the Neurospaces Model Container abstracts model description from the solver. Another component, the Discrete Event System can handle distribution and queuing of spikes. Components adhere to the CBI simulator architecture.

It is possible to develop a MUSIC adapter consistent with the CBI simulator architecture. This would allow the Neurospaces framework, and Genesis 3, to interface to independently running applications in a cluster environment.

Chapter 2

Execution Model

2.1 Phases of Execution

A multi-simulation, i.e. a set of interconnected parallel applications, is executed in three distinct phases:

Launch is the phase where all the applications are started on the processors. During this phase, MUSIC is responsible for distributing and launching the application binaries on the set of MPI processes allocated to the MUSIC job. Since MPI can be initialized first when the applications have been launched, most of this work needs to be performed outside of MPI. In particular, the tasks of accessing the command line argument of the MUSIC launch utility and of determining the ranks of processes before MPI initialization therefore has to be handled separately for different MPI implementations.

Technically, the launch phase begins when `mpirun` launches the MUSIC binary and ends when the Setup object constructor returns. (See further description below.)

Setup is the phase when all applications can publish what ports they are prepared to handle along with the time step they will use and where data will be present (where in memory and/or on what processor). During the setup phase, applications can read configuration parameters communicated via the common configuration file. At the end of the setup phase, MUSIC will establish all connections.

The setup phase begins when the Setup object has been created and ends when the runtime object constructor returns.

Runtime is the phase when simulation data is actually transferred between applications. Via `tick` calls the simulated time of applications is kept in order.

The runtime phase begins when the runtime object has been created and ends when its finalize method is called.

From the application programmers point of view, these phases are clearly separated through the use of two main components of the MUSIC interface:

the *Setup* and the *Runtime* object. The launch phase is not visible for the application since it handles the situation before the application starts.

When the application initializes MUSIC at the beginning of execution it receives a specific *Setup object*. This object gives access to the functionality relevant during the setup phase via its methods. When done with the setup, the application program makes the transition to the runtime phase by passing the Setup object as an argument to the *runtime object* constructor which destroys the Setup object. The runtime object provides methods relevant during the runtime phase of execution.

2.2 Spatial Distribution of Data

Communication between applications is handled by ports. Ports are named sources (output ports) or sinks (input ports) of data flows. The data to be communicated may be differently organized in process memory on the receiver side compared to the sender side. The applications may run on different numbers of processes, and, the data may be differently distributed among the sender processes and the receiver processes, as is shown in Figure 2.1. How does MUSIC know which data to send where?

In MUSIC, there are two views of the data to be communicated over a connection. Data elements are enumerated differently according to these views. MUSIC uses *shared global indices* to enumerate the entire set of data to be sent over the connection while *local indices* enumerate the subset of data which is stored in the memory of a particular MPI process. Data does not need to be ordered in the same way according to the two views. For example, data stored in an array may be associated with an arbitrary subset of global indices in an arbitrary order.

The MUSIC library is informed about the relationship between global and local indices and how data is stored in memory during the setup phase. Two abstractions are used to carry this information:

The `IndexMap` maps local indices to global indices. That is, the `IndexMap` tells which parts of a distributed data array are handled by the local process and how the data elements are locally ordered.

The `DataMap` encapsulates how a port accesses its data. The `DataMap` contains an `IndexMap`. While an index map is a mapping between two kinds of indices, the data map also contains information about where in memory data resides, how it is structured, and, the type of the data elements. The type is used for marshalling when running on a heterogeneous cluster.

During setup every process of the application individually provides the port with a `DataMap` (or an `IndexMap` in the case of event ports).

> **Rationale:** While connections are often used to handle the transfer of spikes from one group of neurons to another, they should not be regarded as an implementation of synaptic projections. Connections will only handle a direct one-to-one transport from one application to another. Re-mapping to actual receiving neurons, e.g. to implement an all-to-all projection, must be handled by one of the applications. Thus, it may be better to regard the ports as *proxy-objects*, providing indirect access to neurons simulated by the other application.

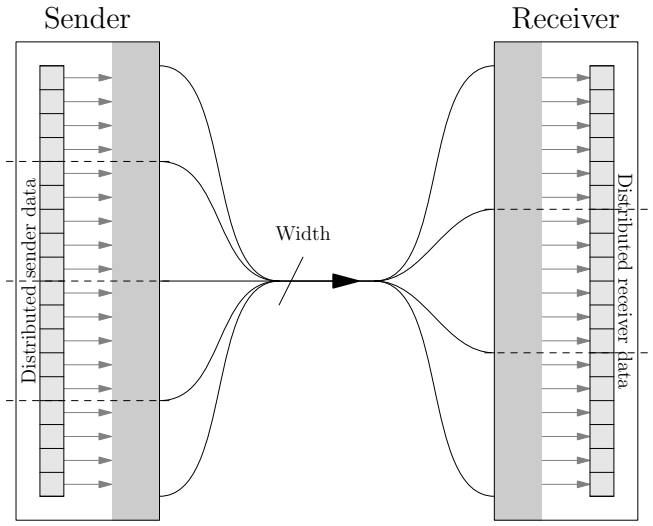

Figure 2.1: Data transfer over a connection from an application running in four processes to an application running in three processes. The light gray areas in the sender and receiver represents the MUSIC port. Dashed lines divide the application into distinct processes.

2.3 Timing Considerations

Different applications may use different time steps and it is the responsibility of MUSIC to ensure that data is delivered at the appropriate time. In order to minimize handshaking, both parts of a connection pair locally calculate when the actual data transfer over MPI takes place. To ensure that these calculations produce predictable results, simulation time is internally represented using integers with a global micro-timestep common for all applications.

Simulation time is local for each application and MUSIC does not enforce unnecessary synchronization between these local clocks. Thus, an application producing data may be running ahead of another application which consumes the same data. MUSIC internally builds a schedule which ensures that data arrives at the appropriate local time in the receiving application. Scheduling becomes more complex when data is not only transferred in a feed-forward manner, i.e. when the connection graph contains loops. In this case MUSIC has to rely on the existence of sufficient delays in the simulated model, typically corresponding to axonal delays.

Figures 2.2 and 2.3 illustrate how MUSIC handles time when transferring continuous data over a connection. In figure 2.2, the sender application uses a shorter tick interval than the receiver. The sender side uses values sampled at the tick points to interpolate a value corresponding to the point in time when the receiver makes its tick call.

The dark middle area (labelled "MPI") is where the actual data transfer takes place. MUSIC makes use of the fact that the receiving application can run with its simulation clock set independently of the sender. The arrows going "backwards in time" in this area reflect the fact that the receivers clock is

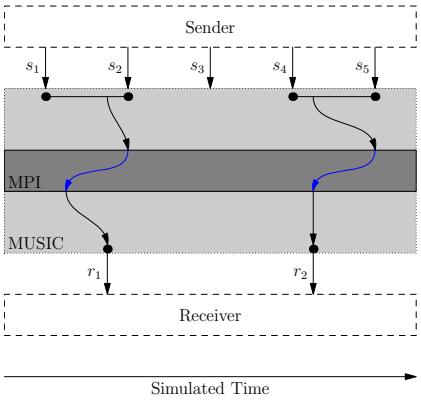

Figure 2.2: Transfer of data when sender has a shorter tick interval than the receiver.

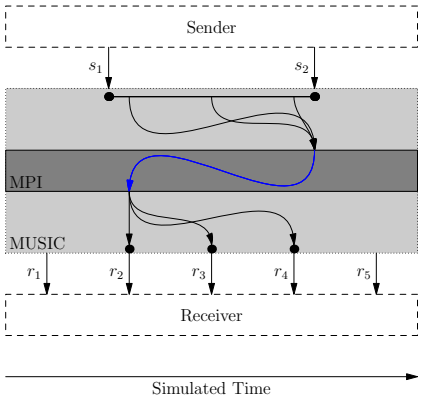

Figure 2.3: Transfer of data when sender has a longer tick interval than the receiver.

lagging. This makes it possible for data to arrive in time despite the fact that it was available later (e.g. at tick s_2) than when it was arriving (at r_1), when talking about simulated time.

Figure 2.3 illustrates what happens when the receiver of continuous data is calling tick faster then the sender. The sender will then buffer up values from the preceding and current ticks and transfer this at a suitable tick call. The receiver will portion these values out by interpolating at the appropriate ticks.

The strategy of having the receiver application running with a delayed local clock only works when the connection graph forms a directed acyclic graph (DAG). When loops occur it is necessary to allow for data arriving late, at least somewhere along each loop. MUSIC handles this via *acceptable latency* which is a property of event input ports. The receiving application declares how late, according to simulation time, data may arrive, thus giving MUSIC room for resolving the scheduling problem. In the case of continuous data, the application specifies a *delay* which fulfills the same purpose.

In figures 2.2 and 2.3, the sending application must be running ahead of the receiver in order to maintain the illusion that communication is instantaneous. Figure 2.4 illustrates the timing relation between sender and receiver along a real time axis (wallclock time) when the receiver accepts a delay of incoming data. This allows the receiver (B) to run ahead of the sender (A), thus creating the slack necessary to make schedules for communication loops.

2.4 Message Ports

In addition to the port types which handle continuous and spike event data, MUSIC provides *message ports*. Message ports allow for transmission of arbitrary messages of, for example, control information between applications. A multi-simulation may, for example, be controlled by a script running in a Python process on one of the cluster nodes. The script may use a message port to alter a parameter or turn on a stimulus in an application at a certain point in time.

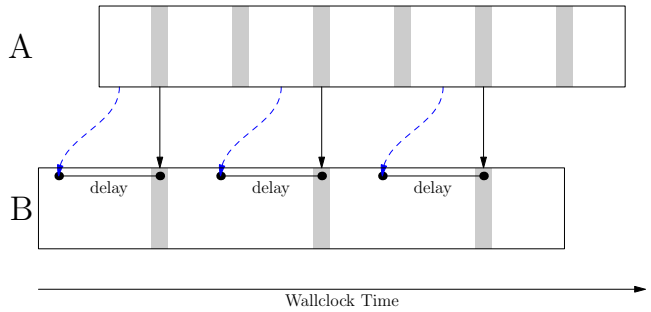

Figure 2.4: This figure illustrates how MUSIC can allow one application (B) to execute ahead of another (A) when transferring continuous data. The receiver (B) has specified a delay on the input port which means that the value to be delivered at each tick (gray areas) corresponds to a simulated time in A (blue arrows) which has already happened.
Note that the tick times when MUSIC actually transfers data will be aligned on the real time axis, since blocking communication is used. In practice, one of the applications will have to wait for the other to reach the same point in its execution.

Messages sent from any process on the sender side of the connection are routed to all processes on the receiver side which have announced there willingness to receive messages. Contrary to ports for continuous data, any marshalling is the responsibility of the sender and receiver application.

To achieve independence between MUSIC applications, it is recommended that messages are text strings with the syntax of the interpreter language of the receiving application, and that these text strings originate from a user-specified configuration file read by the sending application.

2.5 Application Responsibilities

One goal of MUSIC has been to limit the responsibilities imposed on each application (c.f. section 1.2.2). Here we present a step-by-step list of what an application must do in order to participate in a multi-simulation.

1. **Initiate MUSIC**

   This is done by calling the `Setup` function.

2. **Create ports**

   Data to be imported and exported is identified by creating named ports.

3. **Map ports**

   MUSIC is informed about where the actual data is located. This includes information about which processor owns each data element. For continuous data it also includes information about where in memory it is stored, while for event data it specifies how to receive events.

4. **Initiate the runtime phase**

   At this stage, MUSIC can build the plan for communication between different processes.

5. **Advance simulation time**

   The application must call `tick` at regular intervals to give MUSIC the opportunity to transfer data.

6. **Finalize MUSIC**

   By calling `finalize`, all MUSIC communication is terminated.

Chapter 3

Starting a Multi-Simulation

3.1 Overview

Parallel programs based on MPI are normally started by running a special program called `mpirun` (for MPI-1) or `mpiexec` (for MPI-2). To start multiple applications and enable them to communicate with each other, MUSIC utilizes a special launcher program called `music` which, in turn, starts the different applications. Information about which applications should be started, and the communication pattern between them is described in a common *configuration file*.

> **Rationale:** The reason for not controlling configuration via the MUSIC API is that individual applications should remain ignorant about the structure of the full multi-simulation. Thus, the API provides methods for asking about the parts of the configuration relevant for that application, i.e. its ports, but does not expose the complete communication graph.

3.2 The Configuration File

The main purpose of the configuration file is to control what applications to start, and to connect output ports to input ports. The configuration file specifies the number of MPI processes allocated to each application.

The configuration file consists of a sequence of blocks, each starting with a non-indented bracket:

[*application_label*]

Each block consists of a sequence of configuration variable definitions applying to one application. The application label is used to refer to ports of the application. A variable definition takes the form of an assignment:

varname = *value*

The following variable names have special meaning to MUSIC:

binary Pathname to application binary

args Command line given to the binary

np The number of MPI processes to allocate for the application

timebase The length of a MUSIC micro-step, that is, the resolution of MUSIC:s internal clocks). (Default value is 1 ns.)

> **Rationale:** The possibility to specify the MUSIC timebase is provided since the timebase is a compromise between resolution and maximal simulation time. With 64-bit clocks, a timebase of 1 ns gives a maximal simulation time of 585 years which should be sufficient for most applications.

Arbitrarily named parameters may also be included in the configuration file and these parameters can be accessed from the applications.

A connection between an output and input port is specified using the following syntax:

application_label.port_name −> *application_label.port_name*

The direction of the arrow (−>, <−) indicates the direction of data transport. An output port can be connected to multiple input ports while an input port can be connected to, at most, one output port.

Optionally, the width of the connections between applications can be specified:

application_label.port_name −> *application_label.port_name* [*width*]

The application label can be omitted if it refers to the application being specified by the surrounding block. An example of a simple configuration file can be seen in section A.1. Appendix C specifies the formal syntax of configuration files.

> **Rationale:** Information from the configuration file needs to be available both in order to launch the application binaries and during the setup phase. Since launching must be done prior to MPI initialization, it is not possible to distribute configuration information via MPI itself. In the reference implementation of MUSIC, environment variables are used to distribute this information to the applications.
>
> This information transfer is hidden within MUSIC, so a different implementation of MUSIC may use another technique. In particular, if the applications are launched from a scripting program, such as PyNN, that program must also take care of transferring the relevant configuration information to the applications.

Chapter 4

Application Program Interface

4.1 Conventions

This chapter describes the API to the MUSIC library. The API is object oriented and all communication with the library is performed via instance methods of different classes of objects. Appendix B presents an alternative C interface. The most common way of passing objects as arguments in MUSIC is via pointers. The only exception is the Setup constructor. The convention regarding memory management is that the caller should make sure that objects exist in memory during the entire execution of the method, and is also responsible for the deallocation of objects afterwards.

4.2 Error handling

MUSIC attempts to fall back on the error handling mechanisms of MPI. A MUSIC exception thus results in a call to the MPI error handler. A particular implementation of the MUSIC library does not guarantee that it handles all kinds of errors that may occur during MUSIC calls. Each error handled by MUSIC generates an exception, and MUSIC installs suitable error codes, classes and strings so that the MPI error handler is able to generate suitable error messages.

MUSIC follows the style of error handling in the MPI standard, which is described in sections 7.2 and 7.3 in the MPI 1.1 report[1] and in section 2.8 of the MPI 2.0 report[2]. The default error handler of MPI is `MPI_ERRORS_ARE_FATAL` which means that any error handled by MUSIC will result in the program being aborted. Using the error handling of MPI requires features only described in the MPI 2.0 report. For MPI implementations which lack this support, MUSIC uses its own error handler which has the same behavior as `MPI_ERRORS_ARE_FATAL`.

> **Rationale:** MUSIC adheres to the error handling strategy of MPI since the application is already using MPI and should not need to implement a second error handling strategy when converted to use MUSIC.

4.3 Setup

4.3.1 The setup constructor

Each application initializes the MUSIC library through a call to the Setup constructor. This constructor, in turn, calls `MPI::Init` to initialize MPI. The Setup constructor creates the Setup object through which the application can retrieve configuration information, get an application wide communicator, and setup ports.

`Setup::Setup (int& argc, char**& argv)`

`argc`	reference to the `argc` argument of `main`
`argv`	reference to the `argv` argument of `main`

This constructor must be called at most once; subsequent calls are erroneous. It accepts the `argc` and `argv` that are provided by the arguments to `main`.

Example 4.1: Initializing MUSIC

```
int main (int argc, char *argv[])
{
  MUSIC::Setup* setup = new MUSIC::Setup (argc, argv);

  /* parse arguments */
  /* rest of program */
}
```

> **Rationale:** The idea behind creating a specific setup object is to ensure that the application does not accidentally call functions relevant only for the setup phase at other times.

4.3.2 Communicators

During a multi-simulation the MUSIC library will create a unique intra-communicator over the group of processes assigned to each application. This application wide communicator takes the role of the global communicator `MPI::COMM_WORLD` and is retrieved from the setup object through a call to the `communicator` method.

`MPI::Intracomm Setup::communicator ()`

return value	the application wide communicator

The application is supposed to use the application wide communicator in place of `MPI::COMM_WORLD`. If the application binary has been launched using `mpirun` instead of the `music` utility, `communicator ()` will return `MPI::COMM_WORLD` as the application wide communicator.

Example 4.2: Accessing the application-wide communicator

```
/* communicator with global scope */
extern MPI_Comm comm;

...
{
  ...
  comm = setup->communicator ();
  int rank = comm.Get_rank ();
  ...
}
```

Rationale: An alternative to provide the `communicator` function would have been to redefine `MPI::COMM_WORLD`. This would ensure that an application does not accidentally use the global communicator. However, it may not always be possible to dynamically redefine this variable in all MPI implementations, so for the sake of portability, we have chosen a more straightforward technique.

4.3.3 Port creation

Ports are named sources (output ports) or sinks (input ports) of data flows. Output and input ports are distinct classes. Ports are further subdivided into distinct classes depending on whether they handle continuous data, event data or messages.

```
ContOutputPort*
Setup::publishContOutput (string id)

ContInputPort*
Setup::publishContInput (string id)

EventOutputPort*
Setup::publishEventOutput (string id)

EventInputPort*
Setup::publishEventInput (string id)

MessageOutputPort*
Setup::publishMessageOutput (string id)

MessageInputPort*
Setup::publishMessageInput (string id)
```

`id`	port name
return value	an unmapped port

Ports have two stages in life: the *unmapped* stage and the *mapped* stage. A port is unmapped when created. The MUSIC configuration file specifies connections between ports. It is possible to ask an unmapped port if it is

connected, if it has a width specified and, if so, what width it has. A port becomes mapped when its method `map` is called.

Example 4.3: Creating an unmapped port

```
ContOutputPort* out =
   setup->publishContOutput ("out");
```

4.3.4 General port methods

The port API includes methods to ask a port if it is connected, if it has a width specified, and, if so, what that width is.

Port connectivity

The method `isConnected` is used to check if the user has specified a connection of this port to another port in the configuration file.

---

bool `Port::`**isConnected** ()

return value **true** only if connected

This method is typically used in cases where the use of some of the ports of the application is optional. In such a case, it is not sensible to allocate any application resources to support the data flow in question. One example is if one wants to support output of membrane potentials from a certain population of cells, but don't want to waste resources if no one is listening.

Example 4.4: Optional handling of ports

```
ContOutputPort* out =
   setup->publishContOutput ("Vm");
/* map port only if anyone is listening */
if (out->isConnected ())
  /* allocate application resources and map port */
```

Port width

The width of a port, that is the number of data elements transferred in parallel from a cont port or the largest possible id of an event port + 1, can be specified in the configuration file. This should be thought of as a request for a given width. Applications can use the method `hasWidth` to determine if a width has been specified and retrieve it using `width`. Message ports do not have width.

---

bool `Port::`**hasWidth** ()

return value **true** only if port width has been specified

---

int `Port::`**width** ()

return value port width

Rationale: Applications can use the above methods to adapt their port width. A typical usage would be a general purpose post-processing tool which receives information from an ongoing simulation. Such a tool can publish a number of optional input ports and then use `isConnected` and `width` to adapt its internal processing depending on what kind of data source it is actually connected to. See example 4.5.

Example 4.5: Publishing port of adaptive width

```
{
  ...
  /* Publishing a port of adaptive width */
  double* stateVars;
  MUSIC::ContInputPort* in =
     setup->publishContInput ("in");
  if (!in->hasWidth ())
    /* report error */
  else
    {
      int size = in->width ();
      /* for clarity we assume that nElements
         is a multiple of size */
      int nLocal = nElements / size;
      /* example continues as in next example */
      ...

    }
}
```

4.3.5 Mapping cont ports

A port is informed about what data exists locally and how to access it by mapping it. Cont ports transfer data from or to memory during `tick` calls and need to know the layout of data in memory. In addition, the marshalling (conversion between different bit level representations) performed on heterogeneous clusters requires information about the data type being transferred. This information is captured by the data map argument `dMap`. The `DataMap` type is described in section 4.3.9 below.

```
void ContOutputPort::map (DataMap* dMap,
                          int maxBuffered)

void ContInputPort::map (DataMap* dMap,
                         double delay,
                         int maxBuffered,
                         bool interpolate)
```

`dMap`	the data map associated with the port
`delay`	delay of data arrival in simulation time (s)
`maxBuffered`	maximal amount of data buffered (ticks)
`interpolate`	enable interpolation (boolean)

The optional argument `delay` informs MUSIC of when, according to simulation time, to sample data on the sender side. If enabled, linear interpolation is used to obtain an approximation of the state at this time. The default delay is zero. Delayed continuous data may be used in connectionist networks when modeling brain pathways. A delay is *required* at some point when communicating continuous data in a loop (c.f. section 2.3).

Buffering data in output and input ports gives more efficient communication since data can be sent fewer times in larger packets. By default MUSIC buffers some reasonable amount of data. In certain situations it is necessary to be careful about memory usage. Using the optional argument `maxBuffered` the application can give MUSIC a bound on how much data to buffer. MUSIC decides how much data to buffer based on the lowest `maxBuffered` parameter given when mapping each of a set of connected ports and on latency considerations when applications are connected in loops. A `maxBuffered` value of N ticks means: don't buffer more data than is sufficient for communicating at every N th tick.

When the optional argument `interpolate` is **true**, MUSIC uses linear interpolation to determine the values delivered on the receiver side. This is the default behavior. By passing **false** this interpolation can be switched off in which case MUSIC selects the sample on the sender side which is closest according to simulation time.

Example 4.6: Mapping ports to internal data

```
{
  ...
  int size = comm.Get_size ();
  int rank = comm.Get_rank ();
  /* for clarity we assume that nElements
     is a multiple of size */
  int nLocal = nElements / size;
  double* stateVars = new double[nLocal];
  MUSIC::ContInputPort* out =
     setup->publishContOutput ("out");
  MUSIC::ArrayData dMap (stateVars, MPI::DOUBLE,
                         rank * nLocal, nLocal);
  out->map (&dMap);
  ...
}
```

4.3.6 Mapping event ports

```
void EventOutputPort::map (IndexMap* indices,
                           Index::Type type,
                           int maxBuffered)

void EventInputPort::map (IndexMap* indices,
                          EventHandler* handleEvent,
                          double accLatency,
                          int maxBuffered)
```

`indices`	the index map associated with the port
`type`	the indexing type used (local or global)
`handleEvent`	a user-defined event handler
`accLatency`	acceptable latency for incoming data (s)
`maxBuffered`	maximal amount of data buffered (ticks)

Since event ports don't access data the same way as cont ports, they do not require a full `DataMap`. Instead, an `IndexMap` is used to describe how indices in the application should be mapped to the shared global indices common for sender and receiver. The applications has the choice of using local indices or bypassing the index transformation by directly using the shared global indices when labelling events. This is controlled by the `type` parameter which can be set to `MUSIC::Index::LOCAL` or `MUSIC::Index::GLOBAL`.

Events are communicated to the application through an *event handler*. The event handler is called by MUSIC when the application calls `tick`. It is called once for every event delivered.

Some spiking neural network models include axonal delays. The MUSIC framework assumes that handling and delivery of delayed spikes occurs on the receiver side. In such a case, the receiver may allow MUSIC to deliver a spike event later than its time stamp according to local time. The maximal acceptable latency is specified through the `accLatency` argument.

The optional argument `maxBuffered` has a similar meaning as for cont ports above but the actual amount of data buffered is, in this case, not deterministic since it is dependent on spike rate.

Sending events

The sender registers events for transmission by calling the method `insertEvent`.

```
void EventOutputPort::insertEvent (double t,
                                   LocalIndex id)
void EventOutputPort::insertEvent (double t,
                                   GlobalIndex id)
```

`t`	trigger time of the event (s)
`id`	the local or global index of the event

MUSIC guarantees that this event will be delivered through a call to the user-specified `EventHandler` on the receiver side no later that the acceptable latency relative to receiver local time. The time `t` must be between the simulation time of the last tick and that of the next.

The parameter `id` should be converted from **int** to `LocalIndex` or `GlobalIndex` to indicate what kind of indices are used in the application.

Receiving events

```
class EventHandlerLocalIndex {
public:
  virtual void operator () (double t,
                            LocalIndex id) = 0;
};

class EventHandlerLocalIndex {
public:
  virtual void operator () (double t,
                            GlobalIndex id) = 0;
};
```

`t`	trigger time of the event (s)
`id`	local or global index if the event

Event handlers are called by event input ports to deliver events. The application is supposed to customize either `LocalEventHandler` or `GlobalEventHandler` by subclassing one of them (depending on the indexing scheme the application uses).

4.3.7 Mapping message ports

Message ports behave similarly to event ports in that messages are sent and delivered using similar mechanisms, but while events are routed between processes based on event indices, messages are routed to all processes on the receiver side which have provided a `MessageHandler` to `map`. All arguments to `map` for message ports are optional.

```
void MessageOutputPort::map (int maxBuffered)

void MessageInputPort::map (MessageHandler* handler,
                            double accLatency,
                            int maxBuffered)
```

`handler`	a user-defined message handler
`accLatency`	acceptable latency for incoming data (s)
`maxBuffered`	maximal amount of data buffered (ticks)

Sending messages

The sender registers a message for transmission by calling the method `insertMessage`.

```
void MessageOutputPort::insertMessage (double t,
                                       void* msg,
                                       size_t size)
```

`t`	time stamp (s)
`msg`	pointer to message
`size`	size of message in bytes

MUSIC will deliver this message through a call to the user-specified `MessageHandler` on the receiver side no later than `accLatency` with regard to the time stamp.

Example 4.7: Sending a message

```
{
  ...
  char m[] = "string to send";
  port->insertMessage (runtime->time (), m, sizeof (m));
  ...
}
```

Receiving messages

```
class MessageHandler {
public:
  virtual void operator () (double t,
                            void* msg,
                            size_t size) = 0;
};
```

`t`	time stamp supplied by sender (s)
`msg`	pointer to message subclass instance
`size`	size of message instance in bytes

Message handlers are called by message input ports to deliver messages. The application is supposed to customize `MessageHandler` by subclassing. It is recommended that messages are text strings with the syntax of the interpreter language of the receiving application, and that these text strings originate from a user-specified configuration file read by the sending application.

The message given to the `MessageHandler` is deallocated by the MUSIC library.

4.3.8 Index maps

An `IndexMap` is a mapping from the local data element indices to shared global indices. An index map instance thus holds information of which subset of the shared global indices belong to the local MPI process and of their local order. MUSIC implements two subclasses of `IndexMap`: `PermutationIndex` and `LinearIndex`. The most general form is `PermutationIndex` which allows for an arbitrary mapping.

```
PermutationIndex::PermutationIndex (int* indices,
                                    int size)
```

`indices`	vector of shared indices
`size`	number of shared indices

```
LinearIndex::LinearIndex (int baseIndex, int size)
```

`baseIndex`	shared index corresponding to local index zero
`size`	number of contiguous indices in this process

When a cont output port is mapped it becomes associated with a set of state variables (or other data) in the memory of the sender. When the receiver calls `runtime::tick`, an estimate of the values of these variables are stored in a set of variables associated with an input port on the receiver side. Similarly, an event output port is mapped to a set of event id:s.

While the number of variables or id:s on the receiver side is always the same as on the sender side, the data can be distributed in different ways between MPI processes on the sender side compared to the receiver side. In fact, sender and receiver may consist of different numbers of processes.

Index maps are used in each MPI process to tell MUSIC how data is distributed and ordered by enumerating the shared indices represented by the process in local order.

4.3.9 Data maps

A `DataMap` encapsulates how a port accesses its data. While an index map is a mapping between two kinds of indices, the data map also contains information about where in memory data resides, how it is structured, and, the type of the data elements. `ArrayData` is a subclass of `DataMap` which describes arrays of data elements. See example 4.6.

```
ArrayData::ArrayData (void* buffer, MPI_Datatype type,
                      IndexMap* map)
```

`buffer`	data memory location
`type`	data type
`map`	index map

Since data organized in arrays is common, MUSIC provides a convenience form of the array data map constructor which also creates a linear index map:

```
ArrayData::ArrayData (void* buffer,
                      MPI_Datatype type,
                      int  baseIndex,
                      int size)
```

`buffer`	data memory location
`type`	data type
`baseIndex`	shared index of first local element
`size`	number of contiguous indices in this process

4.3.10 Configuration variables

The values of all variables defined in the configuration file can be queried using the method `config`.

```
bool Setup::config (string name, string* result)

bool Setup::config (string name, int* result)

bool Setup::config (string name, double* result)
```

`name`	variable name
`result`	pointer to location where result should go
return value	true if value of correct type was found

Querying for a value of type **int** or **double** expects a value of the correct type, if defined in the configuration file. If the variable is defined, but its value can't be translated into the correct type this causes an error condition.

Example 4.8: Querying configuration variables

```
/* Retrieving the parameter gKCa
   from configuration file */
double gKCa;
if (!config ("gKCa", &gKCa))
  gKCa = 29.5e-9; // default value
```

4.4 Runtime

4.4.1 The runtime constructor

`Runtime::Runtime (Setup* s, double h)`

`s`	pointer to the Setup object
`h`	simulated time increment at each tick (s)

Creation of the runtime object marks the transition from the setup to the runtime phase. The runtime object constructor destroys the Setup object, effectively making it impossible to create new ports. All data structures which have been associated with ports during mapping must be initialized to some suitable start value at the time of the call to the runtime constructor. These values are used during early data transfers of data sampled at negative values of simulation time and, thus, not available.

Example 4.9: Runtime

```
...
MUSIC::Runtime runtime = MUSIC::Runtime (setup, stepSize);
...
```

Rationale: The step size is given as a real number (in seconds) since this makes most sense from the applications point of view. Internally, this number is converted to an integer (using the time micro step time base). This is made to ensure that all processes use exactly the same numbers even when the multi-simulation is running on mixed architectures. Both sides of a connection must agree on when data is transferred over the MPI connector to minimize the need for handshaking during the runtime phase.

Rationale: In order to create a deterministic schedule for buffering and data transfer, we require that `tick` increments simulated time by a fixed amount each time. We realize that some applications may use a variable time step for their numerical integrations, which may then make it harder to execute these tick calls at the right time. However, allowing variable tick steps would have made it impossible to use a pre-computed deterministic schedule and enforced repeated handshaking throughout the runtime phase, resulting in a substantial performance degradation.

Note that the tick step does not need to be equal to the internally used integration step. We believe that most large scale parallel simulators already have some means for fixed interval operations, e.g. to handle logging to files or graphics, which may be utilized also for the tick calls.

4.4.2 The tick

```
void Runtime::tick ()
```

The tick function must be called at regular intervals in simulation time. The application chooses the interval as a parameter to the `Runtime` constructor, normally based on the time step used in the application. The `tick` function is typically called in the main simulation loop of each application. Different applications may use different tick intervals and MUSIC will ensure that time is incremented consistently throughout the multi-simulation.

Before `tick` is called, the application must ensure that all data mapped for output is valid. At the `tick` call, time is incremented and data mapped for input is updated to reflect the new time. Further, installed event handlers will be called *during* the `tick` call to deliver events.

The MUSIC library may, or may not, exchange data with other applications at the tick call. The application must ensure that exported data values are valid when `tick` is called. It must also expect that imported values may change and that event and message handlers are called.

Rationale: The idea behind the `tick` call is to hide the complexity of data buffering and MPI transfer from the application. For efficient data transfer, MUSIC will try to buffer data both at the sending and receiving port in order to send data in large chunks. Internally, MUSIC will use a pre-computed schedule to keep track of at what ticks the actual data transfer should take place and when data should instead be buffered for later transfer.

4.4.3 Simulation time

The method `time` returns local time in seconds.

```
double Runtime::time ()
```

return value local time (s)

`time` returns the local time. Time starts at 0 s and is incremented at every `tick` call. While it is possible, and recommended, to let MUSIC keep track of time for the application, this is not required.

Rationale: To schedule data transfers, MUSIC needs to keep track of the simulation time of all applications via its internal integer representation. If the application independently manages its own clock, typically by incrementing a floating point variable, there is a risk

for drift between the two time representations. The `time` function makes it possible for the application to keep its clock in perfect synchronization with time in the other applications.

4.4.4 Finalization

An application supporting MUSIC should replace its call to `MPI::Finalize` with a call to `MUSIC::finalize`.

```
void Runtime::finalize ()
```

`MUSIC::finalize` makes sure all internally buffered data is sent and finally calls `MPI::Finalize`. Note that this means that communication via MPI will not be possible afterwards.

Chapter 5

Adapting Existing Applications

In this chapter we will highlight the steps necessary to adapt an existing neural simulator to MUSIC. We will assume that the simulator is already using MPI to simulate large networks of interconnected neurons.

The two main tasks that need to be handled are: firstly, to create and map ports for data to be imported and exported, and, secondly, to ensure that the `tick` function is called at regular intervals.

5.1 Creating and Mapping Ports

The application needs to inform MUSIC about what data to import and export, and where this data resides. A simulator will typically use some sort of script files where the user specifies the model and other aspects of the simulation. If possible, it is desirable to extend the scripting language of the simulator so that the user can specify what model variables to communicate, and what names to use for ports.

Assuming that we have introduced such constructs into the scripting language, we must decide on a suitable point in the initialization process where ports should be created and mapped. Since continuous data is read from, or written to, application memory space, the program must have allocated its runtime data structures in order to perform the mapping.

Communication of spikes will use event ports. Function calls are used to send and receive individual spike events. Sending of spikes is relatively straightforward, since the only thing needed is to add a call to the method `insertEvent` at the location where spikes are normally detected in the program. Receiving spikes requires more administration, since the spikes can be received earlier than when they should reach their destination compartment. It is therefore necessary to save incoming spikes in some sort of sorted buffer (typically a priority queue).

In addition, MUSIC will always present the spikes as they appear in the sending group of neurons. In most situations, the receiving application will want to implement a remapping to the target compartments, as illustrated in figure 5.1. One spike may thus end up at multiple postsynaptic compartments, spread out over the processors of the receiving application.

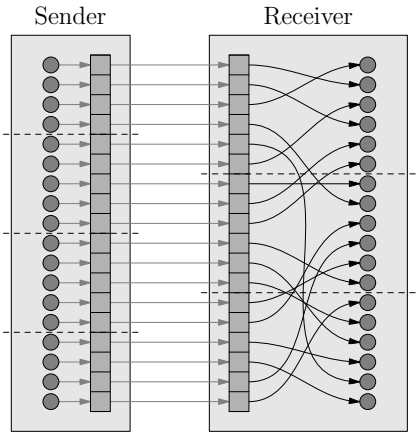

Figure 5.1: The sender application presents the data to the output port in the same order as it is stored internally. The receiving application will see the transferred data in the same order and will explicitly have to implement a proper reordering to implement a typical synaptic projection.

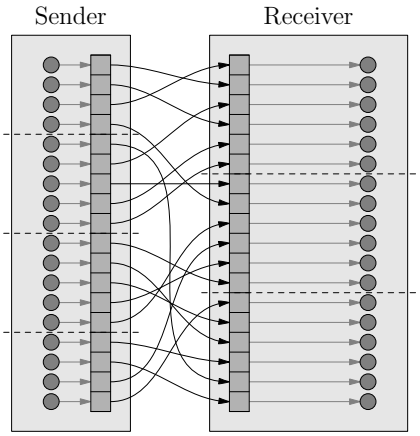

Figure 5.2: If there is a one-to-one correspondence between sending and receiving neurons, the receiving application can specify an appropriate index map to instruct MUSIC to send the data directly to the right destination.

In some situations it may be desirable for the receiving application to avoid this remapping. The application can then utilize the different forms of mappings available in MUSIC to create a general permutation so that MUSIC will send spikes directly to the processor where they should be handled. This situation is illustrated in figure 5.2

5.2 Advancing Simulation Time

The application must call the `tick` function repeatedly throughout the simulation. The application will have to ensure that these calls are made at regular intervals, as specified to the runtime constructor. Note that this refers to *simulated time*; there is no need to consider how much computation time ("wall clock time") is used between tick calls.

If the application makes use of variable time steps internally, it may be necessary to use some sort of checkpoints at fixed intervals where tick can be called. It is not necessary to call tick at every integration time step, but the calls should not be too infrequent.

The tick calls are the only times during runtime when MUSIC will use MPI. MUSIC will then use its own communicators, not to interfere with the MPI operations of the application. Still, we recommend that the application does not intersperse the tick calls with ongoing MPI operations.

When sending continuous values the application must ensure that data arrays mapped for output are filled with values relevant for the time of the *next*

tick. After the `tick` call, data arrays mapped for input will be filled with imported data belonging to the same point in time. Note that `Runtime::time` is updated *during* the `tick` call to reflect the current simulation time.

5.3 Initialization and Finalization

5.3.1 Initiate MUSIC

The idea here is to replace the call to `MPI::Init` with a call to the `MUSIC:Setup` constructor. The Setup constructor calls `MPI::Init` for the application.

The application will have to replace all uses of the global communicator `MPI::COMM_WORLD` with the communicator supplied by MUSIC. The global communicator will be global over all applications and it is necessary to limit the MPI operations to the group of MPI processes belonging to the application.

There should be no need to link an application differently when it is used together with other applications in a MUSIC setting compared to when it is used in a stand-alone setting. In order to support "standard" operation for the application, `Setup::communicator` (), therefore, will return `MPI::COMM_WORLD` if the job is started directly with `mpirun` instead of with the MUSIC launcher.

5.3.2 Initiate the runtime phase

Creating the runtime object will implicitly call the `Setup` object destructor to ensure that the application will no longer be able to change the communication pattern. At this stage, MUSIC can build the plan for communication between different processes.

5.3.3 Finalize MUSIC

The application should also replace its call to `MPI::Finalize`, normally used to shut down communication, by a call to `MUSIC::finalize`. This will internally call `MPI::Finalize` after having flushed all pending data from internal buffers.

Bibliography

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, November 2003.

[2] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`, November 2003.

Appendix A

A Complete Example

This chapter shows a minimal but still complete example. It consists of two applications, `waveproducer` and `waveconsumer`, and a configuration file used to launch and connect them.

A.1 Configuration File

The configuration file starts the waveproducer application on four processors and waveconsumer on three.

```
stoptime=1.0
[producer]
  binary=./waveproducer
  args=120
  np=4
[consumer]
  binary=./waveconsumer
  args=dumpfile
  np=3
  producer.wavedata -> wavedata[120]
```

A.2 Data Generating Application

```
#include <mpi.h>
#include <cstdlib>
#include <cmath>
#include <music.hh>

#define TIMESTEP 0.001

MPI::Intracomm comm;
double* data;

int
```

```
main (int argc, char* argv[])
{
  MUSIC::Setup* setup = new MUSIC::Setup (argc, argv);

  int width = atoi (argv[1]); // command line arg gives width

  MUSIC::ContOutputPort* wavedata =
    setup->publishContOutput ("wavedata");

  comm = setup->communicator ();
  int nProcesses = comm.Get_size (); // how many processes are there?
  int rank = comm.Get_rank ();       // which process am I?

  // For clarity, assume that width is a multiple of n_processes
  int nLocalVars = width / nProcesses;
  data = new double[nLocalVars];
  for (int i = 0; i < nLocalVars; ++i)
    data[i] = 0.0;

  // Declare what data we have to export
  MUSIC::ArrayData dmap (data,
                         MPI::DOUBLE,
                         rank * nLocalVars,
                         nLocalVars);
  wavedata->map (&dmap);

  double stoptime;
  setup->config ("stoptime", &stoptime);

  MUSIC::Runtime* runtime = new MUSIC::Runtime (setup, TIMESTEP);

  for (; runtime->time () < stoptime; runtime->tick ())
    {
      if (rank == 0)
        {
          // Generate original data on master node
          int i;
          double time = runtime->time ();

          for (i = 0; i < nLocalVars; ++i)
            data[i] = sin (2 * M_PI * time * i);
        }

      // Broadcast these data out to all nodes
      comm.Bcast (data, nLocalVars, MPI::DOUBLE, 0);
    }

  runtime->finalize ();

  delete runtime;
```

```
  return 0;
}
```

A.3 Data Consuming Application

```
#include <mpi.h>
#include <music.hh>
#include <fstream>
#include <sstream>

#define TIMESTEP 0.0005

MPI::Intracomm comm;
double* data;

int
main (int args, char* argv[])
{
  MUSIC::Setup* setup = new MUSIC::Setup (args, argv);

  MUSIC::ContInputPort* wavedata =
    setup->publishContInput ("wavedata");

  comm = setup->communicator ();
  int nProcesses = comm.Get_size (); // how many processes are there?
  int rank = comm.Get_rank ();       // which process am I?
  int width = 0;
  if (wavedata->hasWidth ())
    width = wavedata->width ();
  else
    comm.Abort (1);

  // For clarity, assume that width is a multiple of n_processes
  int nLocalVars = width / nProcesses;
  data = new double[nLocalVars];
  std::ostringstream filename;
  filename << argv[1] << rank << ".out";
  std::ofstream file (filename.str ().data ());

  // Declare where in memory to put data
  MUSIC::ArrayData dmap (data,
                         MPI::DOUBLE,
                         rank * nLocalVars,
                         nLocalVars);
  wavedata->map (&dmap);

  double stoptime;
  setup->config ("stoptime", &stoptime);
```

```
  MUSIC::Runtime* runtime = new MUSIC::Runtime (setup, TIMESTEP);

  for (; runtime->time () < stoptime; runtime->tick ())
    {
      // Dump to file
      for (int i = 0; i < nLocalVars; ++i)
        file << data[i] << ' ';
      file << std::endl;
    }

  runtime->finalize ();

  delete runtime;

  return 0;
}
```

Appendix B

C Interface

Most elements of the C interface can be constructed from their C++ counterparts using a few translation rules:

1. All identifiers have the prefix `MUSIC_`.
2. Constructors translate to `create` followed by the class name.
3. Destructors translate to `destroy` followed by the class name.
4. Methods translate to class name followed by method name.
5. References translate to pointers.
6. Strings translate to **char** ∗.
7. Optional C++ arguments are required in C.

Entries which do not strictly follow these rules are preceded with an extra comment in the following listing.

```
/* Setup */

MUSIC_Setup *MUSIC_createSetup (int *argc, char ***argv);

/* Communicators */

MPI_Intracomm MUSIC_Setup_communicator (MUSIC_setup *setup);

/* Port creation */

MUSIC_ContOutputPort *MUSIC_publishContOutput (MUSIC_setup *setup,
                                               char *id);
MUSIC_ContInputPort *MUSIC_publishContInput (MUSIC_setup *setup,
                                             char *id);
MUSIC_EventOutputPort *MUSIC_publishEventOutput (MUSIC_setup *setup,
                                                 char *id);
MUSIC_EventInputPort *MUSIC_publishEventInput (MUSIC_setup *setup,
                                               char *id);
```

```
MUSIC_MessageOutputPort *MUSIC_publishMessageOutput (MUSIC_setup *setup,
                                                     char *id);
MUSIC_MessageInputPort *MUSIC_publishMessageInput (MUSIC_setup *setup,
                                                   char *id);

void MUSIC_destroyContOutput (MUSIC_ContOutputPort* port);
void MUSIC_destroyContInput (MUSIC_ContInputPort* port);
void MUSIC_destroyEventOutput (MUSIC_EventOutputPort* port);
void MUSIC_destroyEventInput (MUSIC_EventInputPort* port);
void MUSIC_destroyMessageOutput (MUSIC_MessageOutputPort* port);
void MUSIC_destroyMessageInput (MUSIC_MessageInputPort* port);

/* General port methods */

/* Xxx = Cont | Event
   Ddd = Output | Input */

int MUSIC_XxxDddPort_isConnected (XxxDddPort *port);
int MUSIC_MessageDddPort_isConnected (MessageDddPort *port);
int MUSIC_XxxDddPort_hasWidth (XxxDddPort *port);
int MUSIC_XxxDddPort_width (XxxDddPort *port);

/* Mapping */

/* No arguments are optional. */

void MUSIC_ContOutputPort_map (MUSIC_ContOutputPort *port,
                               MUSIC_ContData *dMap,
                               int maxBuffered);

void MUSIC_ContInputPort_map (MUSIC_ContInputPort *port,
                              MUSIC_ContData *dMap,
                              double delay,
                              int maxBuffered,
                              int interpolate);

void MUSIC_EventOutputPort_map (MUSIC_EventOutputPort *port,
                                MUSIC_IndexMap *indices,
                                int maxBuffered);

typedef void MUSIC_EventHandler (double t, int id);

void MUSIC_EventInputPort_map (MUSIC_EventInputPort *port,
                               MUSIC_IndexMap *indices,
                               MUSIC_EventHandler *handleEvent,
                               double accLatency,
                               int maxBuffered);

void MUSIC_MessageOutputPort_map (MUSIC_MessageOutputPort *port,
                                  int maxBuffered);
```

```
typedef void MUSIC_MessageHandler (double t, void *msg, size_t size);

void MUSIC_MessageInputPort_map (MUSIC_MessageInputPort *port,
                                 MUSIC_MessageHandler *handleMessage,
                                 double accLatency,
                                 int maxBuffered);

/* Index maps */

MUSIC_PermutationIndex *MUSIC_createPermutationIndex (int *indices,
                                                      int size);

void MUSIC_destroyPermutationIndex (MUSIC_PermutationIndex *index);

MUSIC_LinearIndex *MUSIC_createLinearIndex (int baseIndex,
                                            int size);

void MUSIC_destroyLinearIndex (MUSIC_LinearIndex *index);

/* Data maps */

/* Exception: The map argument can take any type of index map. */

MUSIC_ArrayData *MUSIC_createArrayData (void *buffer,
                                        MPI_Datatype type,
                                        void *map);

/* Exception: MUSIC_createLinearArrayData corresponds to
   c++ music::ArrayData::ArrayData (..., ..., ..., ...) */

MUSIC_ArrayData *MUSIC_createLinearArrayData (void *buffer,
                                              MPI_Datatype type,
                                              int baseIndex,
                                              int size);

void MUSIC_destroyArrayData (MUSIC_ArrayData *arrayData);

/* Configuration variables */

/* Exceptions: Result is char *
   Extra maxlen argument prevents buffer overflow.
   Result is terminated by \0 unless longer than maxlen - 1 */

int MUSIC_config (MUSIC_Setup *setup,
                  char *name,
                  char *result,
                  size_t maxLen);

int MUSIC_config (MUSIC_Setup *setup, char *name, int *result);
```

```
int MUSIC_config (MUSIC_Setup *setup, char *name, double *result);
/* Runtime */
MUSIC_Runtime *MUSIC_createRuntime (MUSIC_Setup *setup, double h);
void MUSIC_tick (MUSIC_Runtime *runtime);
double MUSIC_time (MUSIC_Runtime *runtime);
/* Finalization */
void MUSIC_destroyRuntime (MUSIC_Runtime *runtime);
```

Appendix C

Specification File Syntax

<simulation spec>	::=	{ <application block> }
<application block>	::=	<newline> '[' <application id> ']' { <declaration> }
<application id>	::=	<symbol>
<declaration>	::=	<variable def> | <connection>
<variable def>	::=	<variable> '=' <value>
<variable>	::=	<symbol>
<value>	::=	<integer> | <float> | <string>
<connection>	::=	<port id> <direction> <port id> [<width>]
<port id>	::=	<application id> '.' <port> | <port>
<port>	::=	<symbol>
<direction>	::=	− > | < −
<width>	::=	'[' <integer> ']'

Index