Mace + Distributed Systems HOWTO

James W. Anderson
jwanderson@cs.ucsd.edu

Charles Killian
ckillian@cs.ucsd.edu

Amin Vahdat
vahdat@cs.ucsd.edu

November 20, 2006

Abstract

Mace is a programming language and toolkit for building robust and high-performance distributed systems. The Mace programming language provides constructs to specify services, protocols, and distributed systems as a composition of services, messages, and C++ code. The Mace toolkit provides implementations for many classes of services, including routing, overlay routing, multicast, aggregation, HTTP, and distributed hash tables. The Mace toolkit also includes libraries for encryption, string processing, file I/O, serialization, and XML-RPC.

Contents

1 Preamble

1.1 Copyrights and Distribution

This document is released under "the new BSD license" (listed below). The Mace software package is distributed under the same license restrictions. This manual contains short example programs ("the Software"). Permission is hereby granted, free of charge, to any person obtaining a copy of the Software, to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the conditions of the license:

1.2 Acknowledgments

We thank Adolfo Rodriguez and Dejan Kostic for making Mace possible by developing the original system for building overlay networks, MACEDON. We also thank all the other contributors who have worked on Mace or MACEDON, including Ryan Braud, John Fisher-Ogden, Calvin Hubble, Duy Nguyen, Justin Burke, David Oppenheimer, Sooraj Bhat, and Amin Vahdat. We thank as well in advance all reviewers of this document and those who give feedback.

1.3 Feedback

Please send feedback and corrections to Charles Killian (ckillian@cs.ucsd.edu), James W. Anderson (jwanderson@cs.ucsd.edu). We will be grateful for corrections and will incorporate them as soon as possible.

2 Why are you reading this document?

Perhaps you stumbled on this document on the web somewhere, perhaps you have already downloaded Mace and are trying to figure out how to use it, perhaps you are considering whether to use Mace, or perhaps you've been told to use Mace by a boss or instructor, and need to figure out what that means.

This document will explain what Mace is, go through a simple example of how to use it, and discuss some of its advanced features as well as address common concerns and questions.

2.1 What is Mace?

If you found this document looking for information on various forms of bludgeoning weapons or self-defense sprays, then this paragraph will be all that is in this document about your search. None of these are the context in which Mace will be discussed in this document. The name Mace is instead a derivative from the MACEDON parent project, abbreviated to suggest the broader scope to which it is applicable.

Mace is a software package for building distributed systems. It builds upon the ideas from its parent project, MACEDON, by broadening the scope of what can be designed with it, and by removing many limitations of the original system.

Mace includes a compiler that translates service specifications into C++ code, libraries designed to be linked together with generated services, a distribution of existing services ready to be used by other services or applications, and a few basic applications to run the services contained within.

Mace seeks to transform the way distributed systems are built by providing designers with a simple method for writing complex but correct and efficient implementations of distributed systems. To that end, we are always considering new libraries and language features which could be used to make building, designing, debugging, or verifying distributed systems more powerful, flexible, simple, or natural.

2.2 Why would I want to use a domain-specific language and toolkit?

Over the past few years, we have set out to evaluate a variety of techniques for building robust, high-performance distributed systems. One of our explicit goals has been to conduct head-to-head comparisons of algorithms proposed in the literature, including our own. Target systems include application-layer multicast, distributed hash trees, peer-to-peer indexing systems, and overlay routing. One of the primary lessons of this work thus far has been the difficulty of translating elegant and seemingly simple distributed algorithms into operational, high-performance, and robust systems.

We found significant commonality among the various implementations—for instance, event handling, timer management, failure detection, asynchronous communication, node join/departure, and message serialization/deserialization. However, differences in individual application structure and implementation forced us to repeatedly reimplement the same functionality, typically with a different set of errors and inefficiencies each time. Even once the baseline system was operational, the resulting application often performed significantly slower than expected and bugs in corner cases would remain dormant or masked for long periods of time.

We concluded that a significant impediment to rapid prototyping of our target applications was the lack of an appropriate development environment for distributed applications. Based on our experience, we identified the following requirements to ease this development effort.

Constructing distributed systems would be simplified by the ability to compose simple distributed computing primitives into more complex behavior. For instance, many distributed applications would benefit from failure detection, consensus, multicast, barriers, and key-based routing. However, without well-defined API's, it is difficult to reuse implementations or to leverage the benefits of an improved implementation of a given logical subsystem. In this paper, we describe initial efforts to define required API's to support complex, multi-layer distributed systems.

Current programming languages are not well suited to the requirements of distributed systems. While there are communication libraries and class hierarchies in languages such as C++, Java, and Python, they typically target client/server communications (e.g., HTTP or XMLRPC) and still provide relatively primitive support for failure detection and recovery. Further, we observe that the higher-level structure of many distributed systems is logically event- and state-based. Each node maintains some state that may be modified as a result of a series of events, typically message reception and timer expiration. Individual nodes respond to events by modifying their state and perhaps transmitting their own message to one or more destinations. While this high level structure is simple to describe, it is error prone to implement. Further, managing asynchrony still remains a challenge. Delivering high performance often requires careful consideration of appropriate locking primitives, ensuring that individual operations do not block, and assigning the appropriate number of threads to handle logically concurrent tasks. Of course, all of this can be programmed in existing languages such as C++ and, to a lesser extent, Java. Providing the appropriate language primitives can both significantly simplify the code and reduce opportunities for errors.

Debugging support for distributed systems remains primitive, often reducing to custom scripts run over log files generated at hundreds of individual sites. Generating the necessary debugging information can make the underlying code less efficient, less readable, and more error prone. With appropriate language support, the appropriate logging information can be generated semi-automatically. Further, the information can be efficiently stored in a format amenable to inserting into a SQL database. We have found this log of per-event system activity along with separate programmer annotations for their *expectations* of how the distributed system should behave and *assertions* about global correctness conditions (both of which must necessarily simultaneously apply to state stored at multiple nodes across at multiple time granularities) to be invaluable in debugging distributed system behavior. These distributed expectations and assertions may apply to to both system performance and distributed system structure.

We are building Mace to be just the support that a designers and developers need when building distributed systems. It is a complete redesign and rewrite of its parent project, MACEDON, targeted at a broader range of distributed systems, and with better support for the programmer with fewer limitations.

2.3 Why shouldn't I just use other previously existing languages?

Though there are a few existing languages with similar goals, most do not have the emphasis on building real systems as Mace does. MACEDON of course is very close, but suffers from limitations such as fixed size message headers, restricted and monolithic API, non-portable network communications, lack of support for connecting with external processes (such as XML-RPC or HTTP), lack of support for firewall traversal, more primitive logging, and a single linear protocol/service stack.

(Ed. Note: Comments on other systems forthcoming).

3 Getting and Installing Mace

3.1 Mace dependencies

Mace requires the following system software packages:

gcc/g++ The GNU C and C++ compilers. Mace officially supports GCC version 3.4, 4.0, and 4.1. It has in the past worked with a few GCC version 3.3.X systems, but most frequently causes internal compiler segmentation faults. Using at least version 3.4 is highly recommended. This should also be accompanied with GNU Make.

perl Perl version 5.8 or greater. Additionally, Class-MakeMethods and Parse-RecDescent modules are needed.

system libraries libpthread, libm, libcrypto, libstdc++, and libssl.

libboost Used in a few places for shared pointers and lexical casting.

lgrind For making the documentation from the source files, beautifying the source code. Also needs LaTeX.

Mace has been tested on Debian GNU/Linux and CentOS.

3.2 Getting the Mace source code

Mace is Open Source Software which is be published under a BSD-style license. The latest Mace source code release can be found at the following URL:

http://mace.ucsd.edu/release

Download this file and save it in the directory where you will be doing your development.

To take full advantage of Mace, you should also download the Mace-extras package (includes code licensed under the GPL), available at the same URL.

In particular, a sha1 implementation exists in the mace-extras package, though it is released under a BSD compatible license. Without this code, a simple "hash" will be computed instead.

3.3 Installing Mace

Currently, the Mace distribution does not require the installation of any Mace-related files outside of the Mace source tree. All development using Mace can be done inside the source tree.

To unpack and build the Mace distribution with the mace-extras package, us the following commands (latest is replaced by the version string from the version downloaded):

```
$ tar -xvzf mace-latest.tar.gz
$ tar -xvzf mace-extras-latest.tar.gz
$ mv mace-extras mace
$ cd mace
$ make
```

This will build the Mace libraries (described in § E), services (§ C), and applications (§ F).

4 A Simple Example: Ping

As an introduction to Mace, we present a simple service that checks the liveness of a peer by sending it ping messages. A *service* is a piece of code with a well-defined interface that runs on one or more distributed hosts and provides an implementation of the methods (or services) specified by its interface. For instance, a route service provides methods and an implementation for routing messages to a single host, and a multicast service provides methods and an implementation for delivering a message to many hosts. Services may be used by other services as well as by applications. This HOWTO will show how to write a Ping service and a sample application using the service.

Throughout the rest of this tutorial, all file paths are relative to the `mace` root directory.

4.1 Ping Overview

4.1.1 Usage Model

Our ping application is designed to be run on two hosts. One host is the sender and the other is the responder. The senders sends a message to the responder, who then sends a response message back to the sender. The sender will print that it received a response, or will report an error if it times out.

In § 6, we show how to extend our usage model to support multiple hosts, continuous queries, and more detailed latency reports.

4.1.2 Application and Service Components

Service Class Definition: `PingServiceClass.mh` § 4.2-§ 4.4. The interface for our service, defined in a Mace service class header file.

Handler Definition: `PingDataHandler.mh` § 4.5-§ 4.7. The interface for receiving upcalls[1] from our service, defined in a Mace handler class header file.

Service Implementation: `(First)Ping.m` § 4.8-§ 4.9. The actual implementation of our Ping service, written using the Mace language in a Mace service file.

Application: `(first)ping.cc` § 4.10. The implementation of the ping application. The application 1) instantiates a service instance and a handler instance, 2) registers the handler with the service, and 3) runs the service. This is a standard C++ file/application.

4.2 ServiceClass Definitions

Before implementing the application, we will first design the interface the application will use so that the application and the service may be written independently. The way to do this in Mace is by writing a service class and optionally one or more handlers, which are classes that receive upcalls (*i.e.* callbacks) for event notifications.

All Mace services implement one or more *service classes*, which are specifications of the API provided by the service. Service classes are analogous to *interfaces* in Java. Service classes are defined in Mace header files, with names of the form *ServiceName*`ServiceClass.mh`. Service class header files should be placed in `services/interfaces`. (Mace includes a variety of service classes, which are listed in § B.) We first present our example service class and then describe some of the details (§ 4.4):

4.3 PingServiceClass

The service class definition `services/interfaces/PingServiceClass.mh` follows:

```
serviceclass Ping {
  virtual void monitor(const MaceKey& host);
  virtual uint64_t getLastResponseTime(const MaceKey& host) { return 0; }
  handlers PingData;
}; // PingServiceClass
```

The Ping service provides two methods:

`monitor` Request that the service monitor a new host.

`getLastResponseTime` Query the service for the last time, in epoch microseconds, that a response was received from the given host.

The Ping service also has one upcall handler, of type *PingDataHandler*. Handlers will be described in more detail later.

The type *MaceKey* is a class used to represent the address of a host. For our simple application, these simply represent IPv4 addresses. However, *MaceKey*s can also represent other classes of addresses, such as SHA1 160 bit hashes, which are used by overlay routing protocols such as Chord and Pastry (which is why they are named *keys*).

4.4 ServiceClass Notes

The Mace service header files are very similar to C++ header files. They are, in fact, automatically translated into C++ header files by the build process. Here are some notes to keep in mind when writing service class header files.

`#ifndef ... #define ... #endif` These necessary preprocessor directives to prevent a file from being included twice are automatically generated for the C++ header file. Do not include them in your `.mh` Mace service header file.

[1] upcalls are the Mace lingo for callbacks. They are called upcalls because they are calls from lower levels of services to upper layers

`#include`, declarations, typedefs, etc All code between the start of the `.mh` file and the class declaration is copied verbatim into the generated C++ header file. Put appropriate `#include` directives here.

Class declaration and inheritance All service classes inherit from the base class *ServiceClass*. A ServiceClass may extend other ServiceClasses, but all ServiceClasses have *ServiceClass* as a common ancestor. If no service classes are listed as super classes, then the generated C++ header file will have *ServiceClass* listed as the sole super class. Derived service classes inherit from other service classes using the normal C++ syntax. However, all super service classes must be specified with no modifies (`public`, `virtual`, etc). For instance, the service class declaration for *OverlayRouterServiceClass* is:

```
serviceclass OverlayRouterServiceClass : OverlayServiceClass { ...
```

To define a handler instead of a service class, use the keyword `handler` instead.

Method implementations All service class methods must have an implementation. If desired, a default implementation may be provided in the service header file (as done for `getLastResponseTime`), which will be copied verbatim into the generated C++ header file. If no method implementation is provided, then a default implementation will be generated that will `assert` failure.

`handlers` Mace service header files introduce a new keyword, `handlers`, that declares the handlers used by the service. Recall that handlers are classes (also defined in `.mh` files, described in detail in the next section) that receive upcalls for event notifications. `handlers` should be followed by a comma separated list of one or more handler names, followed by a semicolon. Thus, we are declaring that *PingServiceClass* has one handler, of type *PingDataHandler*.

The `handlers` declaration will cause four methods, `registerHandler`, `registerUniqueHandler`, `unregisterHandler`, and `unregisterUniqueHandler` to be generated for each handler listed. These methods are the means by which actual handler objects are registered and unregistered with the service to receive upcalls when the service has an event to report. For example, the Ping service will report when it receives a response from a monitored host or when it times out waiting for an expected response.

Calling `registerHandler` allows you to register multiple upcall handlers for a single service instance. This is illustrated later in § 6.2. The `registerUniqueHandler` method tells the service that you are registering exactly one upcall handler[2]. We use this in our first Ping application in § 4.10.

Registration IDs In the generated header file, all method signatures will be modified so that they take an additional parameter of type *registration_uid_t* as the last argument that is assigned a default value of *-1*. Registration IDs are not needed by application writers when you register unique handlers. You do not need to worry about registration IDs right now; we mention this for the sake of completeness.

4.5 Handler Definitions

Many Mace services provide the option to register objects to receive upcalls for event notifications (declared by using the `handlers` keyword). To make a method upcall, a service needs an object, with a well-defined interface, on which to make the method call: this object is the *handler*. Handler objects implement a single *handler class* interface. Handler classes are analogous to service classes: they provide API specifications. The APIs for the upcall methods are defined in Mace header files with names of the form *HandlerType*`Handler.mh`. Handler class definition header files should also be placed in `services/interfaces`.

4.6 PingDataHandler

The handler class definition for `services/interfaces/PingDataHandler.mh` follows:

```
handler PingData {
  virtual void hostResponseReceived(const MaceKey& host, uint64_t timeSent,
                                    uint64_t timeReceived, uint64_t remoteTime) { }
```

[2]In truth, it simply registers your handler with a pre-defined and well-known id which only one handler may be associated with.

```
  virtual void hostResponseMissed(const MaceKey& host, uint64_t timeSent) { }
}; // PingDataHandler
```

The Ping data handler provides two methods:

hostResponseReceived Upcall indicating that the given host responded to a ping message. The four parameters are: 1) the host that responded, 2) the time we sent the ping message, 3) the time we received the ping reply, and 4) the time the remote host sent the ping reply.

hostResponseMissed Upcall indicating that the given host failed to respond to a ping message within the timeout period.

4.7 Handler Notes

All the notes for ServiceClass definitions, described in § 4.4, apply for handlers, except for 1) the requirement to extend from a base service class: handlers need not and should not inherit from any super class, and 2) the `handlers` declaration may not be used in handler classes, only service classes.

4.8 Service Implementations

Services are implemented in subdirectories of `services`. Typically, services are implemented in files with names of the form *ServiceName*`.m`, under the subdirectory `services/`*ServiceName*.

4.9 Ping Implementation

We are now ready to implement our Ping service. First, create the new directory `services/Ping`.

The implementation of `services/Ping/FirstPing.m` follows:

```
service FirstPing;

provides Ping;

services {
  Transport router = TcpTransport();
} // services

constants {
  uint64_t PING_TIMEOUT = 2 * 1000 * 1000; // 2 seconds in micros
} // constants

messages {
  Ping { }
  PingReply {
    uint64_t t;
  }
} // messages

state_variables {
  MaceKey target;
  uint64_t timeSent;
  timer sendtimer;
} // state_variables

transitions {
  upcall deliver(const MaceKey& src, const MaceKey& dest, const Ping& msg) {
    downcall_route(src, PingReply(curtime));
  } // deliver Ping

  upcall (src == target) deliver(const MaceKey& src, const MaceKey& dest, const PingReply& msg) {
    sendtimer.cancel();
    upcall_hostResponseReceived(src, timeSent, curtime, msg.t);
  } // deliver PingReply
  downcall monitor(const MaceKey& host) {
    timeSent = curtime;
    target = host;
    downcall_route(target, Ping());
    sendtimer.schedule(PING_TIMEOUT);
  } // monitor

  scheduler sendtimer() {
    upcall_hostResponseMissed(target, timeSent);
  } // expire timer sendtimer
} // transitions
```

For a detailed discussion of the Mace language, see § 5. We will now describe how to compile the service and write a simple application that uses the service.

4.9.1 Compiling the Ping Service

Edit `services/Makefile.services`, and add *Ping* to the list of services specified in the `SERVICES` variable[3]. Then, from `services`, run the command **make**. This will build the subdirectories under `services` specified in the Makefile and will make the library files for each service directory, such as `services/libPing.a`.

4.10 Ping Application

We will now write a simple application that uses the Ping service. Our application will report the round-trip response time latency for live hosts, or will report that the given host is not responding.

Create the new directory `applications/ping` for our program.

We implement our program in `applications/ping/firstping.cc` as follows:

```
#include "SysUtil.h"
#include "Util.h"
#include "PingServiceClass.h"
#include "Ping-init.h"

using namespace std;

class PingResponseHandler : public PingDataHandler {

  void hostResponseReceived(const MaceKey& host, uint64_t timeSent, uint64_t timeReceived,
                            uint64_t remoteTime, registration_uid_t rid) {
    cout << host << " responded in " << (timeReceived - timeSent) << " usec" << endl;
    exit(0);
  } // hostResponseReceived

  void hostResponseMissed(const MaceKey& host, uint64_t timeSent, registration_uid_t rid) {
    const time_t t = timeSent / 1000000;
    cout << "did not receive response from " << host << " pinged at " << ctime(&t);
    exit(0);
  } // hostResponseMissed

}; // PingResponseHandler

int main(int argc, char* argv[]) {
//     Log::autoAdd("Ping::");

  PingResponseHandler prh;

  PingServiceClass& ping = FirstPing_namespace::new_FirstPing_Ping();
  ping.maceInit();
  ping.registerUniqueHandler(prh);
  if (argc > 1) {
```

[3]Note that this is the name of the directory, not the name of the service, so if you used a name other than ‘Ping’ for the directory, you should use that here

```
    ping.monitor(MaceKey(ipv4, argv[1]));
  }

  SysUtil::sleep();

  return 0;
} // main
```

A brief overview of this file:

1. We define *PingResponseHandler*, which inherits from *PingDataHandler*, and overrides both methods.
2. In `main()`, we instantiate a *PingResponseHandler* and a *PingServiceClass*.
3. We call `maceInit()` on our Ping service object, which initializes the service state, and also executes the `downcall maceInit` transition, if defined (we did not define one).
4. We register our *PingResponseHandler* with our service instance as a unique handler. The `registerUniqueHandler` method indicates that there will be only one handler of that type registered with the service. The implication of this is that registration IDs will not be needed to differentiate handlers.
5. When run with no arguments, the program will simply respond to other ping requests with ping replies. As a responder, the program will loop forever in the `select`.
6. When run with a single argument, **ping** will send a single ping message to the given host, and either print the round-trip latency or report that it did not receive a response. The handler callback methods call `exit()`, so the program will then terminate.

Because the Mace runtime libraries use threads, it is important that `main()` does not exit until the services complete their tasks. Thus, we recommend calling `SysUtil::sleep()` with no arguments, which is an efficient way to ensure that the main thread does not cause the process to exit.

4.10.1 Compiling firstping.cc

We need to write a very simple Makefile to build our program. The Makefile `applications/ping/Makefile` follows:

```
APPS = firstping
MACEPATH = ../../

include ../Makefile.common
```

In the directory `applications/ping`, run the command **make**. This will build the binary `firstping` executable.

4.11 Running Ping

Copy the `firstping` executable to another machine. Assuming your two machines are named *alexander* and *philip*, run the following commands.
On *philip*:

```
philip$ ./firstping
```

On *alexander*:

```
alexander$ ./firstping philip
IPV4/philip(192.168.0.2) responded in 503404 usec
```

Now kill the **firstping** process on *philip*. Re-run the command on *alexander*:

```
alexander$ ./firstping philip
did not receive response from IPV4/philip(192.168.0.2) pinged at Mon May  9 22:01:14 2005
```

Congratulations! You have written your first Mace service and application.

5 Mace Specification Overview: Ping in Detail

This section explains the various parts of Ping service implementation.

5.1 Header Files

All code between the start of a `.(m|mac|mace)` file and the first recognized Mace keyword is copied verbatim as C++ code into the generated header file in your service implementation. Thus, you should include whatever standard library header files you need at the top of your `.(m|mac|mace)` file.

You do not, however, need to include header files for mace services or handlers that you reference. The Mace compiler will generate include statements for all the service classes and handlers used by your service, so it is not necessary for you to include these yourself. For instance, even though `FirstPing.m` needs to reference the generated `PingServiceClass.h` and `PingDataHandler.h` files, we can omit the `#include` directives for these files because the compiler will generate them for us.

5.2 Service

The beginning of our example includes a “Service” statement.

```
service FirstPing;
```

The service statement specifies the short name of the service. The long name is generated from the short name by appending “Service” to it, to make *PingService*. The build system provided with Mace requires the service line to match the filename. As a shortcut, this statement may be abbreviated as:

```
service;
```

This abbreviated form will continue to work if the filename is changed. Future versions of Mace will feature a different syntax for the service name.

5.3 Provides

The beginning of our example includes a “Provides” statement.

```
provides Ping;
```

The provides statement specifies the service class of our Ping service. This line causes the Mace compiler to read the *PingServiceClass* interface file (which we also wrote), and generate C++ code for the Ping service so that it inherits from *PingServiceClass*.

The syntax of this statement is simple: the keyword `provides`, the name of the service class (with the suffix *ServiceClass* omitted), and a semicolon. Omitting this line causes your service to only provide the “Null” service class, which is defined just as the initialization and exit functions.

5.4 Services

After indicating the interface of the Ping service, we next specify the set of services we use. In the case of Ping, we mainly need a service which can route messages for us over the network to a destination address. This functionality exists as part of the *RouteServiceClass* interface, and there is an included TcpTransport service which implements it.

To tell Mace we want to use this service, we put it in the services block:

```
services {
  Route router = TcpTransport();
} // services
```

This creates a variable named `router`, a reference to an object providing the *Route* service class (defined in *RouteServiceClass*). We also specify that, by default, this should be constructed as a new TCP service.

5.5 The Route Service Class (briefly)

The *Route* service class defines the `route` method for sending data:

```
bool route(const MaceKey& dest, const std::string& msg, registration_uid_t regId);
```

This method is called with a destination address and a buffer of data to send, as well as a registration id. Delivery of data is made to the appropriately registered *ReceiveDataHandler* using its `deliver` method with a `commType` of *COMM_TYPE_UNICAST*:

```
void deliver(const MaceKey& source, const MaceKey& dest, const std::string& msg,
             registration_uid_t regId);
```

We have now covered the basics necessary to get going. The TcpTransport service, which provides this interface, handles the management and connection of sockets as well as properly catching errors for you, isolating you from this burden (you can register a *NetworkErrorHandler* to receive error notifications if necessary). To send a message to a peer, just call `route`. When the message gets to the remote destination, the TCP service calls `deliver` on the registered data handler with the same message buffer.

5.6 Under The Covers

When you specify that you will use a *Route* service class in `FirstPing.m`, the compiler automatically makes the Ping service a *ReceiveDataHandler*, and when a Ping service is initialized, it handles initializing the RouteServiceClass instance it will use, constructing one if necessary, and then registering itself with that RouteServiceClass instance. The `router` variable, defined in the services block, *actually* refers to the registration id obtained from that registration, not to the service instance. To utilize Ping's "router", instead of calling methods directly on it, you call the methods of it through helper functions with the same parameter list but a modified name. For example, to call `route`, you actually call `downcall_route`. The "downcall_" refers to the fact that the router is a service you are using (and therefore lower in the service stack). We saw the use of "downcall_" both in Ping's transitions for sending data and in the `method_remappings` block, described below.

5.7 Constants

Next we define a constant variable which the Ping service will use to specify the maximum time to wait for a response.

```
constants {
  uint64_t PING_TIMEOUT = 2 * 1000 * 1000; // 2 seconds in micros
} // constants
```

Defining constants is as easy as creating a block for them, and then defining them using normal syntax. Note that you do not use the C++ keyword `const` when specifying service constants. Now you can use these constant variables anywhere else—like in transition code, array sizes and template parameters.

5.8 Messages

Now that we have discussed how the calls are mapped to handle messages, we will discuss how you actually define messages. Ping defines two messages, one for the request sent to the peer, and one for the response.

```
messages {
  Ping { }
  PingReply {
    uint64_t t;
  }
} // messages
```

The first message, "Ping", has no data fields associated with it. The `deliver` method already tells us who sent the data, so there is no need to include the sender as a part of the message. The second message, "PingReply", has one field—an unsigned 64-bit integer named "t".

5.9 State Variables

The state variables[4] are like member variables in a class. They are specific to the running instance of a Ping service, and they are accessible anywhere within the service specification. For the most part they are defined like normal C++ variables.

```
state_variables {
  MaceKey target;
  uint64_t timeSent;
  timer sendtimer;
} // state_variables
```

For the Ping service, we define three variables, one for the destination address of the ping, one for the time we sent the ping, and one for the timer we use to cancel the ping if it takes too long.

5.10 Transitions

The `transitions` block contain methods that the service executes in response to events. Each method is preceded by a keyword specifying the transition type and an optional state expression . There are three types of transition methods:

`upcall` This method will be called by one of the service instances listed in the `services` block that your service uses. Upcalls are the way lower-layer services signal events to higher-layer services (such as delivering a message). Upcall transitions must be defined by a handler of a service that your service uses.

`downcall` This method represents an API call that can be made on your service from either applications or services that *use* your service. Downcall transitions must be defined by a service class that your service provides.

`scheduler` This method will be called by the central Mace runtime library scheduler in response to a timer event. Scheduler methods are the same as the names of the declared timers. Each timer calls its transition when it expires.

The Ping service defines four transitions: two for delivering messages (one for each message type defined in the `messages` block), one to handle the monitor API call, and one to handle the sendtimer. Each transition is described below.

```
upcall deliver(const MaceKey& src, const MaceKey& dest, const Ping& msg) {
  downcall_route(src, PingReply(curtime));
} // deliver Ping
```

The first transition, an upcall, is called on our service by our *RouteServiceClass* `router` when it receives a *Ping* message. (The prototype for `deliver` is defined in *ReceiveDataHandler*, which is listed as one of the `handlers` for *RouteServiceClass*.) When our Ping service receives a `Ping` message, then we want to send a `PingReply` message in response. We do this by making a `downcall_route` call on our `RouteServiceClass` (recall that the object (`router`) on which the method is being invoked, normally passed as the final argument, is automatically inferred). We pass in the destination for the message, `src`, the host that sent us the *Ping* message; and we pass the message we want to route, a *PingReply*, which is constructed with the current time (in microseconds), which is available throughout Mace services using the variable "curtime".

```
upcall (src == target) deliver(const MaceKey& src, const MaceKey& dest, const PingReply& msg) {
  sendtimer.cancel();
  upcall_hostResponseReceived(src, timeSent, curtime, msg.t);
} // deliver PingReply
```

The second transition, another deliver upcall, defines what action we take when we receive a *PingReply* message. This transition has been guarded by a state expression, which in this case verifies that the source of the *PingReply*

[4]Mace services are modeled as finite automata. Thus, the variables defined in the `state_variables` block record persistent state information about the service.

message is the same as the `target` being pinged. If ping replies are received from erroneous destinations, no action will be taken. In this transition we first cancel our timer, because we have received a response. Then we make an upcall to the application or service using us, to report that we received a response. Recall that we defined `hostResponseReceived` in *PingDataHandler*. Notice that you access message fields exactly as you would class member fields, such as "msg.t".

```
downcall monitor(const MaceKey& host) {
  timeSent = curtime;
  target = host;
  downcall_route(target, Ping());
  sendtimer.schedule(PING_TIMEOUT);
} // monitor
```

The third transition, a downcall, overrides the default method implementation for `monitor` specified in *PingServiceClass*. When an application or service requests that we monitor a host, we do the following: 1) store the current time, so that we can report it when we make a success or failure upcall; 2) store the target, so that we can report it on failure and use it to protect the delivery of erroneous ping replies; 3) send a new *Ping* message to the target, using our *RouteServiceClass* `router`; and 4) schedule our timer to expire after `PING_TIMEOUT` microseconds.

```
scheduler sendtimer() {
  upcall_hostResponseMissed(target, timeSent);
}
```

The final transition, of type scheduler, will fire if the timer expires. Because we schedule the timer when we send a *Ping* message in `monitor`, and cancel it when we receive a *PingReply* response in `deliver`, the timer will only ever fire if we do not receive a response within `PING_TIMEOUT` microseconds. If the timer does fire, then we make an upcall on the application or service using us to report that we missed an expected response. We defined `hostResponseMissed` as the other method in *PingDataHandler*.

6 Ping Revisited

In this section, we revisit our Ping service, expanding it to make it considerably more flexible and to demonstrate more aspects of Mace.

6.1 Revised Ping Implementation

We are going to revise our Ping service to incorporate the following changes:

1. Use UDP packets for sending messages, since the reliable byte-stream interface of TCP is unnecessary.
2. Use a run-time configurable ping interval and timeout.
3. Provide the ability to ping multiple hosts simultaneously.
4. Ping monitored hosts indefinitely (as opposed to sending a single message).
5. Support multiple registered handlers.

The implementation for our revised service `services/Ping/Ping.m` follows:

```
#include "mmultimap.h"
#include "mlist.h"

service;

provides Ping;

services {
  Transport router = UdpTransport();
  //     Transport router = TcpTransport();
} // services

constants {
  uint64_t DEFAULT_PING_INTERVAL = 1*1000*1000; // useconds
  uint64_t DEFAULT_PING_TIMEOUT = (DEFAULT_PING_INTERVAL + 5*100*1000); // useconds
} // constants

constructor_parameters {
  uint64_t PING_INTERVAL = DEFAULT_PING_INTERVAL;
  uint64_t PING_TIMEOUT = DEFAULT_PING_TIMEOUT;
} // constructor_parameters

states {
  ready;
} // states

typedefs {
  typedef mace::list<uint64_t> PingTimeList;
  typedef mace::hash_map<MaceKey, PingTimeList> PingListMap;
  typedef mace::hash_map<MaceKey, uint64_t> TimeMap;
  typedef mace::multimap<MaceKey, registration_uid_t> RegistrationMap;
} // typedefs

method_remappings {
  implements {
    upcalls {
      messageError(MaceKey const & k, TransportError::type code, std::string const & m);
    }
  }
}

messages {
  Ping {
    uint64_t t;
  }
  PingReply {
    uint64_t techo;
    uint64_t t;
  }
} // messages

state_variables {
  PingListMap sentTimes;
  TimeMap responseTimes;
  RegistrationMap rids;
  timer sendtimer __attribute((recur(PING_INTERVAL)));
} // state_variables

transitions {
  downcall (state == init) maceInit() {
    state = ready;
    sendtimer.schedule(PING_INTERVAL);
  } // downcall maceInit

  (state == ready) {
    upcall deliver(const MaceKey& src, const MaceKey& dest, const Ping& msg) {
      downcall_route(src, PingReply(msg.t, curtime));
    } // deliver Ping

    upcall deliver(const MaceKey& src, const MaceKey& dest, const PingReply& msg) {
      if(sentTimes[src].empty() || sentTimes[src].front() > msg.techo) {
        return; // ping already notified as failure
      }
      while(!sentTimes[src].empty() && sentTimes[src].front() <= msg.techo) {
        sentTimes[src].pop_front();
      }
      responseTimes[src] = curtime;
      if(curtime−msg.techo <= PING_TIMEOUT) {
        upcallSuccess(src, msg.techo, responseTimes[src], msg.t);
      } else {
        upcallFailure(src, msg.techo);
      }
    } // deliver PingReply

    downcall monitor(const MaceKey& host, registration_uid_t rid) {
      if (rids.find(host) == rids.end()) {
        ping(host);
      }
      rids.insert(std::make_pair(host, rid));
    } // downcall monitor

    downcall getLastResponseTime(const MaceKey& host) {
      return responseTimes[host];
    } // downcall getLastResponseTime

    upcall messageError(const MaceKey& k, TransportError::type code,
                        const std::string& m) {
      maceout << "timeout to " << k << " (" << code << ") " << Log::toHex(m) << Log::endl;
    }
  } // state == ready

  scheduler sendtimer() {
    for (PingListMap::iterator i = sentTimes.begin(); i != sentTimes.end(); i++) {
      if(!i−>second.empty() && curtime − i−>second.front() > PING_TIMEOUT) {
        upcallFailure(i−>first, i−>second.front());
        while(!i−>second.empty() && curtime − i−>second.front() > PING_TIMEOUT) {
          i−>second.pop_front();
        }
      }
      ping(i−>first);
    }
  }
} // transitions

routines {
  void ping(const MaceKey& host) {
    downcall_route(host, Ping(curtime));
    sentTimes[host].push_back(curtime);
  } // ping

  void upcallSuccess(const MaceKey& k, uint64_t ts, uint64_t tr, uint64_t rt) {
    for (RegistrationMap::iterator i = rids.find(k); i != rids.end() && i−>first == k; i++) {
      upcall_hostResponseReceived(k, ts, tr, rt, i−>second);
    }
  } // upcallSuccess

  void upcallFailure(const MaceKey& k, uint64_t ts) {
    for (RegistrationMap::iterator i = rids.find(k); i != rids.end() && i−>first == k; i++) {
      upcall_hostResponseMissed(k, ts, i−>second);
    }
  } // upcallFailure
} // routines
```

We discuss the details of the new implementation in § 7. Next, we will present the revised Ping application and usage.

Recompile your updated Ping service by running **make** in the `services` directory.

6.2 Revised Ping Application

We now present our revised ping application `applications/ping/ping.cc` implementation:

```
#include "SysUtil.h"
#include "Util.h"
#include "PingServiceClass.h"
#include "Ping-init.h"
#include "TcpTransport-init.h"
#include "UdpTransport-init.h"

using namespace std;

class PingResponseHandler : public PingDataHandler {

  void hostResponseReceived(const MaceKey& host, uint64_t timeSent, uint64_t timeReceived,
                            uint64_t remoteTime, registration_uid_t rid) {
    cout << host << " responded in " << (timeReceived − timeSent) << " usec" << endl;
  } // hostResponseReceived

  void hostResponseMissed(const MaceKey& host, uint64_t timeSent, registration_uid_t rid) {
    const time_t t = timeSent / 1000000;
    cout << "did not receive response from " << host << " pinged at " << ctime(&t);
  } // hostResponseMissed

}; // PingResponseHandler

class ErrorHandler : public NetworkErrorHandler {
public:
  void error(const MaceKey& k, TransportError::type error, const std::string& m,
             registration_uid_t rid) {
    cerr << "received error " << error << " for " << k << ": " << m << endl;
  }

  void messageTimeout(const MaceKey& k, TransportError::type error, const std::string& m,
                      registration_uid_t rid) {
    cerr << "message timeout " << error << " for " << k << ": " << Log::toHex(m) << endl;
  }
}; // ErrorHandler

static bool shutdownPing = false;

void shutdownHandler(int sig) {
  shutdownPing = true;
} // shutdownHandler

int main(int argc, char* argv[]) {
  Log::autoAdd("Ping::");
//    Log::autoAddAll();
//    Log::setLevel(2);

  SysUtil::signal(SIGINT, &shutdownHandler);
  SysUtil::signal(SIGTERM, &shutdownHandler);

  Params::Instance()−>loadparams(argc, argv);

  PingResponseHandler prh;
  ErrorHandler eh;

  BufferedTransportServiceClass& router = TcpTransport_namespace::new_TcpTransport_BufferedTransport(TransportCrypto::NONE, false);

  router.registerHandler(eh);

  PingServiceClass& ping = Ping_namespace::new_Ping_Ping(router, 5*1000*1000, 6*1000*1000);

  ping.maceInit();
  registration_uid_t rid = ping.registerHandler(prh);
  for (int i = 1; i < argc; i++) {
    ping.monitor(MaceKey(ipv4, argv[i]), rid);
  }

  while (!shutdownPing) {
    SysUtil::sleepm(500);
  }

  ping.maceExit();
  return 0;
} // main
```

We have changed the implementation as follows:

1. Define *PingLatencyHandler*, which reports the one-way latency for successful ping messages.
2. Remove calls to `exit()` in *PingResponseHandler*.
3. In `main()`, instantiate a *PingLatencyHandler*.
4. We now pass the values for `PING_INTERVAL` and `PING_TIMEOUT` as constructor parameters when we instantiate the Ping service object, overriding the defaults.
5. We register both handles with our Ping service instance using `registerHandler`, instead of `registerUniqueHandler`, and we assign the resulting registration uids to variables.
6. We loop over the command line arguments, and call `monitor`—once for each handler—with each host listed on the command line.

The program takes zero or more hosts as command line arguments, specified as either host names or IP addresses in the standard numbers-and-dots notation. If any hosts are specified, we have our Ping service to monitor them. If no hosts are specified, then the program will simply respond to ping requests from other hosts.

To compile `ping.cc`, add "ping" to the list of applications specified by `APPS` in `applications/ping/Makefile`:

```
APPS = ping firstping
MACEPATH = ../../

include ../Makefile.common
```

You can then run make to build the new executable `ping`.

Now try running the revised **ping** application. Try passing it multiple hosts on the command line. Begin monitoring a host; then kill the **ping** process, wait a few seconds, and restart it on the monitored host; watch the output on the monitoring host.

7 Mace Specification Revisited

In this section, we will discuss the changes we made for the revised Ping implementation.

7.1 Services (Revised)

We changed the default route service to a UDP *RouteServiceClass*. We do not need the reliable byte stream semantics of TCP, and a UDP route service has less overhead.

```
services {
  Route router = UDP();
//    Route router = TcpTransport();
} // services
```

Note that we do not have to change *any* other code, other than the "= UDP()", to switch between TCP and UDP route services. This illustrates a key advantage of using service class interfaces: because both the *TcpTransport* and *UDPService* provide the *RouteServiceClass*, our code will compile seamlessly with either of them.

7.2 Constructor Parameters

The `constructor_parameters` block allows you to list variables that can be initialized as parameters in the service constructor. These values must be given defaults in the parameter list for the constructor.

```
constructor_parameters {
  uint PING_INTERVAL = DEFAULT_PING_INTERVAL;
  double PING_TIMEOUT = DEFAULT_PING_TIMEOUT;
} // constructor_parameters
```

We make `PING_INTERVAL` and `PING_TIMEOUT` constructor parameters, so that they can be set at runtime, as opposed to being fixed as constants of the Ping service. We also provide defaults, which are defined in the `constants` block.

7.3 States

The `states` block allows you to list the possible states that your service could assume[5]. All services have an implicit state named *init*. You can change the state of your service by assigning the new state (one of those declared in the `states` block) to the special Mace variable `state`, which is defined for each service instance. You can also reference your current state through this variable to compare it to any of the states declared in the `states` block, plus `init`. We will see examples a little later.

```
states {
  ready;
} // states
```

Our Ping service defines just one new state, `ready`. We will use this state to prevent many methods from being called on our service before the `maceInit` method is called. Note that qualifying the maceInit function with the guard expression is not strictly necessary, the generated code treats the `maceInit` specially and only allows it to be called once. This is also true of the `maceExit` function.

7.4 Typedefs

The `typedefs` block allows you to alias types needed for your service with the normal C++ `typedef` command. The primary purpose for this block is to allow you to create typedefs that can both use types defined in the `auto_types` block (not used in this example) in the typedef and use typedefs in that and the `messages` block.

```
typedefs {
  typedef mace::list<uint64_t> PingTimeList;
  typedef mace::hash_map<MaceKey, PingTimeList> PingListMap;
  typedef mace::hash_map<MaceKey, uint64_t> TimeMap;
  typedef mace::multimap<MaceKey, registration_uid_t> RegistrationMap;
} // typedefs
```

Ping defines four new types: *PingTimeList*, which we will use to store a list of times which unacknowledged ping messages were sent; *PingListMap*, which we will use to associate a *PingTimeList* with each host; *TimeMap*, which we will use to store the last time that we received a *PingReply* message from a host; and a *RegistrationMap*, which will allow us to store which handlers (as indicated by their registration id) are interested in callbacks for which hosts.

7.5 Messages (Revised)

Since in our new ping service, we will send multiple pings to each host, it may be necessary to distinguish which one a given reply corresponds to. A simple way of accomplishing this is to pass the time the ping is sent to the recipient and having them echo this back to the sender. Thus we add an additional field to each message.

[5]Recall that services are modeled as State Machines. These states are the high-level states of the finite automata.

```
messages {
  Ping {
    uint64_t t;
  }
  PingReply {
    uint64_t techo;
    uint64_t t;
  }
} // messages
```

7.6 State Variables (Revised)

```
state_variables {
  PingListMap sentTimes;
  TimeMap responseTimes;
  RegistrationMap rids;
  timer sendtimer __attribute((recur(PING_INTERVAL)));
} // state_variables
```

We have changed our state variables to include the three maps, mentioned above. We still have the *timer* `sendtimer`, as before. However, we use optional Mace syntax to specify that it is a recurring timer. That is, the timer will automatically reschedule itself to expire again after `PING_INTERVAL` microseconds.

7.7 Transitions (Revised)

Our transitions have been revised and are somewhat more involved. You will notice that they now make use of the optional state expressions. These can occur either immediately before the transition name, in parentheses, or they can appear as a block around a set of transitions, with the state expression in parentheses preceding the opening of the block. State expressions may be arbitrary boolean expressions, with the restriction that they may not cause side-effects or modify any of the service's state. State expressions may reference any of the service's state variables and global variables (constants, constructor parameters, and `state`). A transition will only be executed if its state expression evaluates to *true*. If no state expression is provided, *(true)* is assumed (meaning of course that the transition should always be executed). However, for any given event, only the first matching transition with a true state expression will be executed.

```
downcall (state == init) maceInit() {
  state = ready;
  sendtimer.schedule(PING_INTERVAL);
} // maceInit
```

Here we define the `maceInit()` API method, overriding the empty default implementation in *ServiceClass*. The state expression `(state == init)` ensures that `maceInit()` will only be executed if our service is in the `init` state. Calling `maceInit()` in any other state (such as `ready`) will result in a *NOP*.

`maceInit()` does two things: 1) it changes the service's state to `ready`, and 2) it schedules the timer. Because the timer is recurring, it only needs to be scheduled once—it will continue to fire every `recur()` interval until it is canceled.

```
(state == ready) {
  ...
} // state == ready
```

This is the second form of state expression syntax, which indicates that all transitions within the enclosing block should only be executed if we are in the `ready` state. This prevents them from being called before `maceInit`.

```
upcall deliver(const MaceKey& src, const MaceKey& dest, const PingReply& msg) {
  if(sentTimes[src].empty() || sentTimes[src].front() > msg.techo) {
    return; // ping already notified as failure
  }
  while(!sentTimes[src].empty() && sentTimes[src].front() <= msg.techo) {
    sentTimes[src].pop_front();
  }
  responseTimes[src] = curtime;
  if(curtime-msg.techo <= PING_TIMEOUT) {
    upcallSuccess(src, msg.techo, responseTimes[src], msg.t);
  } else {
    upcallFailure(src, msg.techo);
  }
} // deliver PingReply
```

The `deliver()` callback for the receipt of a *PingReply* message has the same idea as in our previous Ping service, but it has to do some bookkeeping. First, we check if the ping has already been timed out in the sendtimer transition, and exit if it has. If not, we update the unacknowledged ping list, and then update the response time for the given host. Then, if the ping reply was received before the timeout we use the routine `upcallSuccess()`, defined in the `routines` block (below), to notify interested handlers of the result. Otherwise, we use the `upcallFailure()` routine to notify interested handlers of the negative result.

```
downcall monitor(const MaceKey& host, registration_uid_t rid) {
  if (rids.find(host) == rids.end()) {
    ping(host);
  }
  rids.insert(std::make_pair(host, rid));
} // monitor
```

We update the `monitor()` API method as follows. We check if we are already monitoring this host. If not (the condition checked in the `if`), then call the routine `ping()`. This check is needed to suppress sending duplicate *Ping* messages to a host when different handlers request to monitor the same host. We also need to insert the mapping from the host to the registration id in our registration map, which will allow us to make callbacks on the interested handler.

```
downcall getLastResponseTime(const MaceKey& host) {
  return responseTimes[host];
} // getLastResponseTime
```

We implement the `getLastResponseTime` API method as a straight-forward lookup in our map.

```
scheduler sendtimer() {
  for (PingListMap::iterator i = sentTimes.begin(); i != sentTimes.end(); i++) {
    if(!i->second.empty() && curtime - i->second.front() > PING_TIMEOUT) {
      upcallFailure(i->first, i->second.front());
      while(!i->second.empty() && curtime - i->second.front() > PING_TIMEOUT) {
        i->second.pop_front();
      }
    }
    ping(i->first);
  }
}
```

Because our service monitors multiple hosts, and does so indefinitely, our timer transition has more complexity. The timer is responsible for two main tasks: 1) notifying timeouts (failures) and 2) sending the next *Ping* message. Both of these tasks must be performed on each host that we are monitoring. Thus, we loop over the `sentTimes` map, which returns us a (host, PingTimeList) pair, for every host that we are monitoring.

We need to make a failure callback when the PingTimeList (which is storing the times of unacknowledged pings) is non-empty, and the first element of the list is more than `PING_TIMEOUT` microseconds in the past. After we notify the failure, we remove all unacknowledged ping messages which are expired from the list.

Finally, we send our next ping to the host, regardless of whether we think it is live or not.

7.8 Routines

Routines are simply C++ methods that are not part of the service API (as either downcalls or callbacks). They are defined within the `routines` block. Service routines should be defined for the same reason you would define a normal class method: to eliminate code duplication for common procedures, or to abstract away a complicated piece of code, etc.

```
void ping(const MaceKey& host) {
  downcall_route(host, Ping(curtime));
  sentTimes[host].push_back(curtime);
} // ping
```

The `ping()` method sends a *Ping()* message to the host, and records the time in our sent times map.

```
void upcallSuccess(const MaceKey& k, uint64_t ts, uint64_t tr, uint64_t rt) {
  for (RegistrationMap::iterator i = rids.find(k); i != rids.end() && i->first == k; i++) {
    upcall_hostResponseReceived(k, ts, tr, rt, i->second);
  }
} // upcallSuccess

void upcallFailure(const MaceKey& k, uint64_t ts) {
  for (RegistrationMap::iterator i = rids.find(k); i != rids.end() && i->first == k; i++) {
    upcall_hostResponseMissed(k, ts, i->second);
  }
} // upcallFailure
```

`upcallSuccess()` and `upcallFailure()` are helper methods that ensure that we make the appropriate callback on all interested handlers. In both methods, we loop over our registration map, which contains (host, registration-id) pairs.

8 A Real Example: DHT

In progress....

9 Advanced Features

FUTURE NOTE: This section will eventually be in the context of building a larger service such as the DHT service. For now it just describes many of the different topics which will be covered in context in the future.

9.1 Registration UIDs

Registration UIDs were seen in action in the detailed ping example. Registration UIDs represent the logical link between two services (or more specifically, a service and one or more handlers). So when a tree service uses a routing service, the tree service should use a registration UID to refer to itself when requesting data be sent. This allows the routing service to know the destination *ReceiveDataHandler*. It also allows the hypothetical tree service receiving the data to know the source service variable, in case it uses more than one routing service. The requirement for registration UIDs is that these registration UIDs be unique, but that they are the same across the nodes of the network. The present implementation allows this by assuming parallel code and a fixed load order, where the UIDs are assigned.

When registering handlers for a service variable, *-1* may be passed in as the UID the first time, to which, the assigned registration UID will be returned. This registration UID should be used in all future calls to the same service variable, both for registrations and normal API calls. This is handled automatically for generated services; the service variable itself is a reference to the registration UID. No assumptions should be made about the assignment of UIDs; in the future this may be used like a cookie, and presentation of the cookie will be a loose form of authentication.

Note: Registration UIDs are also addressed in the FAQ (10).

9.2 Service Composition

One of the powerful features of Mace is the ability to leverage services others have written and to add features to them. For example, consider the implementation of a generic tree multicast service. This service makes use of two separate services – one which takes care of maintaining the tree structure, and the other which manages routing data over the network. Since tree services provide a common API to give the tree structure, a single generic service can implement the algorithm to multicast data to parents and children.

Constructing the services for a generic tree service might look like this:

```
services {
  Route data = TcpTransport();
  Tree tree = RandTree();
}
```

This indicates that the GenericTreeMulticast service will work over whatever router and tree are passed in, but that if either service is not passed in to the constructor, new services will be created according to the default listed.

9.3 Local Addresses

Each service exports a *Local Address* to those which use it, in the form of a *MaceKey*. For many services and situations, this will just be the IPV4 address the node is bound to. However, the network address could include a proxy address (where peers should send traffic destined for it), or it could be part of a different address family together. Services like Chord and Pastry use addresses based on SHA1 hashes, and systems like SkipNet use free-form strings.

Any service may be asked for its notion of the local address via the `getLocalAddress()` function in `ServiceClass.h`. Services should use this address from their lower-level services rather than make assumptions, so that they may more generically run over any service providing the right interface. Note, however, that the addressing a service exposes to a higher layer may be different from the addressing of lower services (which may in fact differ from each other). In this fashion Pastry reports a local address based on the SHA1 hash of the local IP address, but uses IP addressing for all internal communication.

For those with a true need to know about the IPV4 address of the local machine, they may use the helper function `Util::getAddr()`. A somewhat better helper function to use is `Util::getMaceAddr()`, which also returns the proxy address set. By default, this does a lookup on the machine's hostname, and returns the address associated with that hostname and the default port. However, for machines which do not have an address associated with their machine's name, for machines which have multiple addresses and you want to control which one is used, or if the default code just isn't working for you, you may set the parameter `MACE_LOCAL_ADDRESS` in the Params class, which should be in the format "hoststring:port" or "hoststring:port/hoststring:port", where the first is the locally bound address and port, and the second appends to that the proxy address (which can be used in conjunction with port forwarding on firewalls, for example).

9.4 Auto Types

Auto types are really just inline data structures with compiler generated features. These features include Serialization for Messages, XmlRpc-serialization for interfacing with remote processes, the ability to dump and print these in automated debugging, and to create "nodes" from them to store and use in *NodeCollections*. While it is true that you could define these data structures in external header files, and write your own `serialize`, `toString`, and other sundry functions, being able to include them in your service is a convenience. Since it is common for services to have some private internal representation of its state, this represents a simple way to do so.

To define an auto type, simply include it in the `auto_types` block, and define it as you would any other data structure[6]. If you want to use the type in a *NodeCollection*, specify that it is a node type in its defaults. The full set of options will be described in the technical manual. For an example, however, consider the following example:

```
auto_types {
  child __attribute((node(fail_detect=enabled; score=delay;))) {
    double delay __attribute((serialize(no)));
  }
}                                                                    5
```

[6]Though without the trailing semicolon

This defines a child class which represents a node, has auto failure detection code generated automatically[7], and which returns the value of the delay field for the default score function. Then, by scoring the round-trip delay in the delay field, you can quickly query a node collection to find out who has the least score – therefore the smallest round-trip latency. Furthermore, the delay field will not be serialized if a child is sent over the network in the message. This can be useful if for example, the type cannot be easily serialized, or if you are trying to conserve bandwidth.

9.5 Message Defaults

When using messages for communication with other nodes[8], it may be desirable to utilize per-message defaults for each call you might make with a message.

Consider this: when developing a new routing protocol, you may want to consider using multiple TCP services (*i.e.* socket connections or application queues) for communication in your service. The point of using multiple TCP services is that each service will maintain an independent queue of messages to send your peers. Therefore, if one is expected to frequently be full due to congestion backup, it is commonly desirable to have another one on hand which you will use more sparingly for more important control messages. You can even set the queue size independently for each TCP service when it is constructed. Alternately, you may have two routing services, one of which you expect will be encrypted, and the other in plain text. In both of these cases, the chances are that although you have multiple ways to route messages, you will have one way that you ”normally” send each type of message. Specifying per-message defaults allows you to set default parameters for fields following the message. In the *RouteServiceClass*, this includes the default *registration_uid_t*[9]. This way, when you actually send a message, unless you override the defaults, what you “normally” want to happen will. Furthermore, if you later change your mind about what you “normally” want to do, you can change it in a single place.

Now, as an example, consider the `RouteServiceClass::route()` call. Without using defaults, in the transitions of the `.(m|mac|mace)` file, you would write the following code to send a *foo* message with no fields to peer “dest” over the “control_” routing service:

```
downcall_route(dest, foo(), control_);
```

But, if you specify the following message defaults when defining foo:

```
messages {
  foo [downcall_route(const MaceKey&, const Message&, registration_uid_t regId = control_);] { }
}
```

Then when you actually want to send it you can just write:

```
downcall_route(dest, foo());
```

Apart from being much shorter, if you later add an encrypted route service to your service, you can change which service the *foo* message is sent over by just changing the message defaults. And you can still, on a case-by-case basis override the defaults.

9.6 Method Remapping Block

Earlier, when writing the Ping service, we glossed quickly over the `method_remappings` block, indicating simply that it is important to make sure that messages are handled properly. The basic issue is that since the Mace compiler generates code based on arbitrary interface header files, it does not know *a priori* what strings mean to services. For example, they could in fact be any of the messages defined in the messages block, or any serializable type, or just a plain old string. By default, the compiler assumes a string is just a string, and relies on the annotations in this block by the service implementor to tell it otherwise. An added effect of putting this annotation in the service description is that different services may interpret and use the strings differently. However, in the future, we may also consider annotating the interface files to allow a different default behavior.

[7] Although failure detection is enabled for this type, no detection will take place unless an instance of a node type or node collection is specified to perform failure detection using a particular route service

[8] And really, what else are you going to use messages for?

[9] And remember that this is how you actually specify which lower-level service to route the message over

A second issue is how to set default parameters for calls made from the service. Calls made with Message parameters have the ability to set per-message defaults, as discussed in the previous section (§ 9.5). But we need to provide a mechanism for setting defaults for non-messaging calls, and to provide defaults for all messages.

In the future, we will also be allowing more flexible remappings, which for example allow you to remap types using anonymous inline functions, and which allow you to remap functions to other functions.

The `method_remappings` block is broken into two parts: `uses` calls and `implements` functions.

Uses Calls

Uses calls are calls which a service writer will make from their service into either services which they use (downcalls), or to handlers which are registered with them (upcalls). This declarations in the block should match the declarations in their respective header files, with the following differences.

function prefix Each function should be prefixed either by `downcall_` or `upcall_`, appropriate to its kind.

optional string handling Any string parameter may have a serialization instruction provided, which can be any type mace knows how to serialize and deserialize. The type, and possibly `const` and/or reference (`&`) should appear in the declaration where the string declaration would normally go. Then, an arrow pointing to the right (`->`) is used to indicate that for this function you will pass in the type you specified, but it should be converted to the type from the original function declaration before performing the downcall. The type modifiers will be applied to the object as it is passed around for serialization or deserialization, and do not necessarily have to match those from the actual function declaration. [10] This is why in the `downcall_route` specification, the specification is *[const Message&]*. Since in these calls, a temporary Message is constructed, the gcc compiler will complain if the Message parameter is non-const. The Message is passed by reference, to avoid unnecessary memory copies before serialization, or to ensure the deserialized string fills in the provided message. Note that "Message" is a special string for serialization, since this will cause the compiler to both apply per-message defaults, and to handle determining the message type on deserialization and calling the appropriate transition with the message delivery specified.

default values Default values may be specified. These are done in the usual fashion by naming the variable and providing a default value for it. After doing so, the parameter can be left out of the call, and the default will be provided.

```
method_remappings {
  uses {
    downcall_route(const MaceKey&, const Message& -> const std::string&, registration_uid_t);
    downcall_joinOverlay(const NodeSet&, const std::string& identity=std::string(), registration_uid_t regId = router_);
    upcall_verifyJoinOverlay(const MaceKey&, const std::string&, registration_uid_t regId = activeAuthenticator);
  }
  //... service-implemented calls will be described next
  implements {
    //... be patient for this one
  }
}
```

In the above example, we're specifying 3 different uses calls with different instructions. In the first case, we're indicating that when we call `downcall_route`, for the second parameter, instead of passing in a string, we will pass in a *const Message&*, and that it should be automatically serialized to a string. Since it is a constant reference, there will be no consideration for the deserialization of the string parameter after the call to `downcall_route`. The second call specifies that on the `downcall_joinOverlay` call, the default identity is the empty string, and the default registration UID is the one assigned to the `router_` service variable (assuming one exists). The third mainly differs in that it is specifying an upcall, and that the regId default should be activeAuthenticator, which may be a state variable rather than a service variable.

[10] Note that the Mace compiler will [soon] correctly handle updating your data item if a non-constant string reference is used (and therefore possibly modified). If you don't want to allow that to happen, you can add the constant modifier to your declaration of your type, which will prevent the Mace compiler from deserializing the string as the function call returns.

Implements

The `implements` sub-block remaps calls which may be made on the service the writer is writing. These include both functions of the API defined by service classes they provide (downcalls), and callback functions/upcalls for handlers they register with services they use (upcalls). Their declaration also closely follows the original declaration from the header file, and in this case no qualification about the type of call is necessary since there are sub-blocks for the different types. Unlike the uses block, the implements block does not support default values since these calls are incoming from elsewhere, they will already have the parameters filled in. However, the syntax for specifying serialization instructions are the same, save for the direction of the arrow being reversed (`<-`) to indicate that the remote caller is passing in a string and it should be deserialized into your type before your transitions are called. The same provisos are true about the `const` and `&` modifiers on the type, and serialization after your transition is done. This will typically be used to specify the handling of strings in upcalls either as messages or as other serializable types.

```
method_remappings {
  uses {
    //... see the prior section
  }
  implements {
    upcalls {
      deliver(const MaceKey&, const MaceKey&, const Message& <- const std::string&,
              registration_uid_t);
    }
    downcalls {
      //... the same syntax can be used here
    }
  }
}
```

In this example there is a single callback specification, and when deliver is called, the 3rd parameter will be handled like a message.

9.7 Automated Failure Detection

Automated failure detection is presently disabled due to incompatible updates to the compiler, but will be fixed soon. Synopsis: After enabling failure detection for a node-auto type, an instance of a state variable (node or node collection) may have failure detection active. Liveness probing will automatically be done to the specified peers, and when failures are detected, they are signaled to the service via special transitions firing.

9.8 Authentication

Authentication will be covered when the DHT is included in the manual. Synopsis: A single authenticator per-group or per-overlay is registered or set at any given point in time (per-node). Local authentication is handled by that active authenticator, and remote authentication goes locally to that authenticator. We are adding identity strings to several of our calls to make this possible.

10 Frequently Asked Questions

10.1 Registration UIDs

I have a question about *registartion_uid_t*(s) ... basically when do I need them and where do I get them from?

Registration UIDs within Mace services are mainly handled by the compiler. When, in a `.(m|mac|mace)` file, you do:

```
downcall_route(dest, msg(some, stuff), router_);
```

`router_` is the registration uid for the Route service named `router_` in the `services` block. Similarly, if you specify that in defaults for the message (e.g.

```
join_msg [downcall_route(const MaceKey&, const Message&, registration_uid_t regId = router_);] {
  //stuff
}
```

`router_` is still the *registration_uid_t*.

However, when doing upcalls from the `.(m|mac|mace)` file to a higher level (i.e. to the application), you need to know the *registration_uid_t* of the higher level.

For example, when the application calls `query(...)` on your Mace service, the last parameter of the query call will be the *registration_uid_t* of the registered *QueryResponseHandler* to which the response should go.

Typically, You can implement this by including the *registration_uid_t* from the query call in the messages and responses. Then, when doing the upcall – pass in that *registration_uid_t* from the message. The supporting code generated by the Mace compiler will take care of looking up the correct handler to make the call to.

From the application – after you create the service, you need to register the query response handler. If you pass in *-1*, the service will return you a registration UID. If you pass in something else, that will be used as the registration UID. If you want to create your own, you can get one from the *NumberGen* class. But the first technique is preferable. Once you have a registration UID in the application – you should use that for all further calls to that service (i.e. future registrations, calls for queries, etc.)

Note: Alternately from the application, you may call `registerUniqueHandler` on the service, as in the first *Ping* application. This has the effect of registering your handler with a default and well known UID. Then, when making future calls, you may omit the last parameter, allowing it the [well known] default value for the UID. Note that this only works if a single application calls `registerUniqueHandler`, as implied by its name (and the fact that only one handler can be registered under any given UID).

10.2 Compiler Errors

TBD

10.3 Compiler Warnings

TBD

10.4 Linker Errors

When I try to link my application, I get a message such as "undefined reference to '*Myservice*_namespace::new_*Myservice*_*ServiceClass*' ". Why can't it find it?

The straightforward answer, and the one most useful if the suggestions below don't help, is that the linker cannot find the library your service is available in. Each service directory is compiled into one or more libraries for of the format *dirname*.*compiletype*.a, depending on the compile type you're using (typically O2 or O0, for optimization levels), assuming you haven't touched the Makefiles for the service directory. When you link your application, you need to make sure that you include each of the service directory libraries you will be using. If more than one service is contained in a single directory, all services will be included in that directory's library. The library itself is generated in the services directory, so you can see if it exists there. If not, you may just need to run **make** at a high enough level to build the services directory. Another problem occurs with the order of the linked directories. **g++** is not smart enough to chase down all dependencies, so you may need to include libraries more than once. In general, you solve this problem by in `services/Makefile.services` by listing the services in order such that future directories depend on earlier directories. The default application makefile then generates the `RSERVICES` variable which is the reverse order, the one appropriate for linking. Recently another problem was observed by a mace user—setting `LIBNAME` in the application makefile to be the same name as a directory in services caused the service directory to not be found, instead linking in the application library twice. So make sure your application `LIBNAME` does not collide with any other libraries you use, including directory names for services.

10.5 MapIterators, NodeCollections, and Iteration

Okay, so I'm using *NodeCollection*(s) of `node`-enabled `auto_type`(s). But how do I iterate over all the elements?

NodeCollection(s) provide two methods of iteration. The first is to use the *MapIterator* interface. Suppose for example that you had an `auto_type` called `child`. Then, in the `typedefs` block, you could have:

```
typedefs {
  typedef NodeCollection<child, MAX_CHILDREN> children;
}
```

This would create a convenient type named `children` to refer to these node collections. Then, in your state variables you could define a children variable like this:

```
state_variables {
  children myKids;
}
```

Then, to iterate over myKids, you could do so using the *MapIterator* either in a `for` loop or a `while` loop. You'll notice, however, that the `for` loop is a bit non-standard in that the incrementation portion of the loop is empty.

For:

```
for(children::map_iterator i = myKids.mapIterator(); i.hasNext(); )
{
  child& kid = i.next();
  maceout << "Now considering: " << kid.getId() << Log::endl;
}
```

While:

```
children::map_iterator i = myKids.mapIterator();
while(i.hasNext())
{
  child& kid = i.next();
  maceout << "Now considering: " << kid.getId() << Log::endl;
}
```

The second technique for iterating over a node collection is to use the *set_iterator* type (as returned by `setBegin()` and `setEnd()`), which works like standard STL iteration, with the proviso that its like iterating over the node collection as a set, not as a map (for an iterator `i`, `*i` is the node type the collection is templated on).

I did iteration just like the first technique you suggested over a collection in a message, but the gnu-c compiler won't compile the code. What's more, the error message is some template stuff I don't understand. Do you know what's wrong? Or, I'm doing STL iteration like I see it being done elsewhere, but things aren't compiling.

You should ask yourself if the collection (i.e. *NodeCollection*, *hash_map*, etc.) is a `const` collection. If so, you'll need to use either the *const_map_iterator* or the *const_iterator*. This will allow you to iterate, but the items you are iterating over will be constant to prevent you from changing them. Also keep in mind that message fields are constant, even if the message itself is not – so anything in a message will automatically be constant.

10.6 downcall_ and upcall_ helper functions

Since I know the API of the services I'm using and the handlers I can make calls to, why can't I just use syntax more like making calls on the object? Why do I have to use "silly" `downcall_` and `upcall_` helper functions?

In theory this could work. However, since the generated code is handling these references internally for you, all you have accessible are the registration UIDs. On a technical note, we want you to use the helper functions, both to be consistent with the necessity to do so when default values are to be used when making calls or when serialization instructions have been given, and because having a helper function gives us (the compiler writers) an ideal place to instrument and insert code to do other useful stuff. One nice thing about this approach is that you can specify default registration UIDs for each call, and then never have to worry about the references or the registration UIDs again.

If the demand is great enough, we have also talked about adding syntax which would let you write code to call functions on registration UIDs, which could get mapped appropriately, but thus far is unsupported.

10.7 64-bit Mace

Can I run Mace unmodified on 64-bit systems?

Short answer, no.

We would like to get this working, but we are researchers rather than full-time developers, and it's enough work keeping everything running on 32-bit systems. Furthermore, we just don't have 64-bit systems to test/run the system on, so we wouldn't be able to test it anyway. We do welcome tips and patches which make progress towards the goal of being 64-bit clean. Additionally, we were recently tipped about a work-around if you want to run Mace and can live with a 32-bit binary on 64-bit machines.

What follows is an *unsupported* set of tips from a user on compiling 32-bit Mace binaries on and for x86_64 systems.

`Makefile.vars` added *-m32* to `GLOBAL_VARS`

`compiler/Makefile` added -m32 to `CFLAGS` and `CXXFLAGS`, and rules for **macecc** and **serviceparser**

`compiler/XmlRpc/Makefile` added -m32 to **xmlrpcc** rule

`application/Makefile.common` added -m32 to the *$(APPS)* rule. Added another -lcrypto to the *LIBS+=rule* of `MACE_EXTRAS_SHA1`.

copied files `/usr/include/openssl/opensslconf-i386.h` to the top mace directory, `/usr/lib/libl.a` to `lib` directory, and `/usr/lib/libcrypto.a` to the `lib` directory.

A few extra comments from the tip:

```
Mace is, unfortunately, not 64-bit safe. But apparently, RHEL 64-bit can
compile and run 32-bit code (although some of the 32-bit libraries are
missing, so you have to grab them from a 32-bit machine).
```

A Directory Structure

This section covers the directory structure of Mace. You should read this section if you're either can never figure out where a file is in the repository, or if you are trying to figure out where to put a new file.

At the top level, there are 5 directories in the release.

lib The lib directory contains general purpose code with useful interfaces that may be useful in building compilers, applications, and services. They include the logging facilities, a facility for operating on command-line and file parameters, the Mace stl extensions, a variety of Util functions and much more. An overview of these is in § E. This directory should contain code which (a) has no dependencies to other Mace directories, and (b) will be generally useful to more than one application, service, or compiler.

compiler The compiler directory contains the source for compilers. The primary compiler is the `macecc` compiler which operates on `.(m|mac|mace)` files and produces C++ source code. This compiler will typically be run from the front-end perl script `mace`, which also runs the Mace pre-processor (`macepp`) on files named with the `.m` or `.mace` extension. Files named simply with the old `.mac` extension are assumed to be pre-processed already and ready for the `macecc` compiler. Another compiler here is the XmlRpc compiler, compiled in the `XmlRpc` subdirectory. It creates RPC interface files from a header file – this process is described more in § H.

services The services directory contains the interfaces (ServiceClass and Handler definitions) for services, as well as the implementation of various services. Services by definition provide some service class, though they may be a mix of hand-crafted (C++ implemented) and generated (via Mace compiler). The interfaces are all stored in the `services/interfaces` directory, without other files there. (A possible exception is considered for files which define things needed by interfaces but for which it does not make sense to place in the `lib` directory.) The services themselves are stored in the remaining subdirectories. Frequently, each directory implements a single service, though it may include supporting files or test code in that directory. Occasionally, however, related services are placed in the same directory. Note that any `*.cc` in a service directory will be included in the directory's library. The included services and interfaces are covered further in § B and § C.

application This directory contains the various applications included with Mace. Some are toy applications but most could be made to do useful work with only small modifications. The applications are described in § F. Additionally, there is a `common` subdirectory which includes code which is to be shared between applications, as building blocks, but which include dependencies in the `services/interfaces` directory, and therefore cannot be included as part of the `lib` directory. Note that to the extent possible, this directory should *not* include dependencies on particular services, in an effort to allow this directory to be built without regard for what services are being built.

docs The files necessary to produce this documentation. It can be produced by running "make". To produce it you will need latex and lgrind installed. For convenience, a pre-built PDF is already included.

B Service Classes

Service classes you will recall define the public interface of a service. Handlers are interfaces of callback or upcall objects registered with a service class.

All service classes derive from the base *ServiceClass*, which defines the initialization and de-initialization routines (`maceInit` and `maceExit`), respectively. These functions are special functions for the Mace compiler: it reference counts them, and recurses them (only once) to the services used. It also ensures that the maceInit or maceExit event only occurs once for the service.

The specific interfaces for each service can best be seen by looking at the `.mh` file in the `services/interfaces` directory – an overview of what they contain will be here.

Sending and Receiving Data *RouteServiceClass*, *MulticastServiceClass*, and *HierarchicalMulticastServiceClass* define the routing functions found in the original MACEDON system. Each takes a *ReceiveDataHandler* registration for delivery of data. The primary addition of the *HierarchicalMulticastServiceClass* is the `collect` and `distribute` calls, which transmit data *up* or *down* the hierarchy, respectively. In the *ReceiveDataHandler* there is a `forward` call which is performed at all nodes processing the data, including source and destination(s), and a `deliver` call, which is called to deliver the data. Also related to these services is the *NetworkErrorHandler*, which is used to report node failures, and the *BandwidthRouteServiceClass*, which provides an interface for measuring the bandwidth to a set of peers. which is achieved by the routing.

Aggregation and Gossip *AggregateServiceClass*, *GossipServiceClass*. and their respective handlers are the Mace interfaces for aggregating data and gossiping data. Each of these interfaces allows a user to register different aggregations or gossip values for different channel ids.

Structures Service classes have been developed for a few common structures (so far trees and overlay routers), to allow shared code for operations over these structures, separating out the actual maintenance of the structure. These service classes are *OverlayRouterServiceClass* and *TreeServiceClass*. A special service class presently exists for *ScribeTreeServiceClass*, which defines the extra interface Scribe-like services provide so they may be managed by SplitStream-like services. This service class will likely have its name changed when it is adapted for more general-purpose use.

Overlay and Group These two service classes define the interface for joining overlays or groups. They are separated because it is believed that the same interface will be useful for different kinds of services, for example joining a multicast group or joining a group for threshold cryptography or the like[11].

Http The Http service class actually provides little interface into it. It's primary value is the ability to register a handler with the Http service which lets you respond to Http queries in Mace-generated code.

Null The Null service class is one which derives directly from *ServiceClass* and provides no additional interface. Its existance is generally for test code, which is supposed to do its task as a result of `maceInit()`.

C Packaged Services

The following services come pre-implemented in Mace (divided by directory located).

Transport Two *RouteServiceClass* services will eventually appear in this directory. The first is already available – the *TcpTransport*. It provides new features as compared with the historical MACEDON TCP transport, as well has having the added benefit of being actively developed and supported. New features include built-in encryption and authentication using SSH agents, and the ability to run Mace instances using different ports and through port-forwarding firewalls. It also stands as the building block for a number of new features we are considering.

TCP *TCP* represents the original MACEDON TCP transport and provides the *RouteServiceClass*. It is being deprecated, as a new transport with additional functionality and a cleaner design is replacing it. It remains in the release now for continuity and historical significance.

UDP *UDP* represents the original MACEDON UDP transport and provides the *RouteServiceClass*. It is being deprecated, just as the TCP service was replaced by the new *TcpTransport*. It provides simple routing over a UDP socket.

ReplayTree *ReplayTree* represents the original MACEDON random tree, and provides the *TreeServiceClass* interface by building a tree randomly. Its random tree features are deprecated by the new *RandTree*, but it is still useful for constructing a tree by specifying its structure in a configuration file (it *replays* a tree determined offline).

RandTree *RandTree* provides the *TreeServiceClass* interface by building a random tree. Every node joins at the root and finds a place in the overlay tree randomly. This implementation, unlike the MACEDON version provides a degree of tolerance for network failures, including root failover.

RanSub This directory contains implementations of *RanSubAggregator*, an implementation of the *AggregateServiceClass*, and *RanSub*, an implementation of the *GossipServiceClass*. Unlike a "standard" gossip service, *RanSub* provides a uniformly random subset of peer data (the gossip) using an aggregation service which works simply by passing up and down the tree.

Pastry This is an implementation of the original Pastry overlay router. It does include some degree of recovery from network failures, though it does not include any of the newer Microsoft work on controlling the cost of reliability. It does provide proximity neighbor selection.

ScribeMS This is an implementation of the Scribe tree protocol. The MS refers to the fact that in MACEDON there were two implementations of Scribe, one which followed more of the Microsoft design, and one which followed more of the Rice design, which have somewhat diverged from the original publication. This release contains several bugfixes as compared to the MACEDON implementation.

SplitStreamMS This is an implementation of the SplitStream forest multicast protocol. The MS refers to the fact that in MACEDON there were two implementations of SplitStream, one which followed more of the Microsoft design, and one which followed more of the Rice design, which have somewhat diverged from the original publication. This release contains several bugfixes as compared to the MACEDON implementation.

[11]Similar concepts to groups are starting to appear for aggregation, gossip (channel id), or file distribution (file name) – this notion will likely be merged in a future release

GenericOverlayRoute This directory contains two implementations of routing over an overlay. *RecursiveOverlayRoute* provides basic overlay routing using recursive lookups (at each hop, consult with an overlay router to find out the next hop). *CacheRecursiveOverlayRoute* uses recursive routing techniques, but establishes direct links to destination nodes for applications such as tree multicast which frequently don't want intermediate nodes forwarding data between two tree-peers. It does this transparently however so that higher level services need not concern them with the physical addresses of the destination nodes but can just refer to them by their overlay identifiers.

GenericTreeMulticast Implements the *HierarchicalMulticastServiceClass* given an instance of a *TreeServiceClass* and a *RouteServiceClass*. The assumption of this service is that the addressing schemes of these two service instances are the same (that nodeids correspond between the two).

Http Provides an implementation of the *HttpServiceClass*, with which you may register to receive URL requests matching a certain path. May be used either as a service by another service, or as a standalone service.

There is effort underway to port services from MACEDON to Mace, and these will be included and described as they available.

D Logging

All log messages have selectors and log levels. Logging can additionally be enabled or disabled at compile time or runtime.

For a log to get printed, it's level must be below (or equal to) the log level of the system. It's selector must be selected, and the logging must not be disabled. Furthermore, there is a way to disable (at compile time) all normal logs by defining some macros and/or setting a compile-time max log level.

For relevant code, look at `lib/mace-macros.h` and `lib/LogIdSet.h`.

Selectors are added by the macro `ADD_SELECTORS()`; If called by `ADD_FUNC_SELECTORS`, it uses `__PRETTY_FUNCTION__`. Mace services generally use *$service::$function::$state* or *$service::$function::$message::$state*. *$message* is a message name or timer name, *$state* is the guard, or "(demux)" if it's the demux method, or some other strings on generated methods. However, this format can be overridden by the `log_selectors{}` block. This can be done to shorten selectors for example. A prefix is added sometimes based on the macro and *LogIdSet*.

All logging macros use log level 0, except `macecompiler` and `macedbg` (or their `printf` variants). These take a required log level, though the `compiler` versions should only be used by generated code.

Finally, there is the configuration of logging output. This handled by

- `Log::add`
- `Log::autoAdd`
- `Log::autoAddAll`
- `Log::configure`

(See `lib/Log.h` for prototypes)

`Log::configure` utilizes those other ones, and references a number of configuration parameters (from the *Params* singleton), including *MACE_LOG_AUTO_SELECTORS*, *MACE_LOG_LEVEL*, and more. The `add`, `autoAdd`, and `autoAddAll` methods allow configuration of which selectors get printed to which `FILE*`(s), and using what formats (timestamp, selector, thread id, etc.). By default, *WARNING* and *ERROR* are `autoAdd`ed to `stderr`. This can be disabled. Selectors can be printed to more than one file descriptor. Different selectors can be printed using different formats to the same file descriptor, and the same selector can be printed using different formats to different file descriptors.

A log selector string is mapped to a `log_id` at the beginning of the execution. If the selector is not selected before the mapping occurs, *-1* is returned. *-1* causes fast-pass log suppression. Otherwise the log id is an index into a data structure telling how to do the logging.

You can in theory call `Log::logXXX("selector"...)`, which checks the selector mapping each time (though once a mapping is selected, I'm not sure it is re-evaluated). But that is highly inefficient – instead the `log_id` should be used whenever possible, which the mace logging macros handle for you.

So that's just a brief introduction to Mace logging. It is supremely flexible and highly optimized, albeit a bit complex to fully understand.

That should help you get started.

E Useful Library Functions

TBD

The `lib` directory contains a large number of useful library functions. These fall into categories such as STL wrappers, Logging, and Utility functions. At present I'll merely advise looking at the `.h` files to see what's available.

F Applications

The following applications (largely toy applications) are presently being released with Mace.

appmacedon `appmacedon` is a port of the original appmacedon toy application. It is used as a driver for multicast and unicast services, and will attempt to stream [bogus] data at a given rate. It will report average bandwidth and latency for each node at the end of the run.

unit_app `unit_app` is an application for performing unit tests. It runs by constructing an instance of a *NullServiceClass*, calling `maceInit()`, sleeping for the duration of the run, and then calling `maceExit()`. In the future it may support a type of return code which could indicate the success or failure of the test.

filecopy `filecopy` is a simple application to transmit a file over a constructed *RouteServiceClass* and store it at the destination. It has primarily been used to measure the performance of the low-level transports.

http TBD

common Okay – not really an application. TBD

G Method Remappings

As specified by the *RouteServiceClass* API, to route a message over the `router`, you would have to pass in a string. You could manually create that string and have it in a certain message format (known as serializing the message) that you could read back later (deserializing it). However, writing this is mechanical and tedious. Thus, Mace will automate this process for you by allowing you to specify messages, which we will cover in § 5.8. For this automatic conversion to work, however, you have to tell the Mace compiler that the strings in the `route` downcall and in the `deliver` upcall should actually be handled as messages. The following code, placed in the `method_remappings` block, will do just that:

```
method_remappings {
  uses {
    downcall_route(const MaceKey&, const Message& –> const std::string&);
  } // uses

  implements {
    upcalls {
      deliver(const MaceKey&, const MaceKey&, const Message& <– const std::string&);
    } // upcalls
  } // implements
} // method_remappings
```

At this point we are going to give a cursory overview of this syntax. At a high level, you are remapping the `route` downcall (which you *use* from the `router`) and the `deliver` *upcall* (which you *implement* for the `router`) to calls which have slightly different parameters – namely using a constant "Message" reference. "Message" here is a keyword to tell the compiler that it could be replaced by any message. The arrow just denotes the direction in which the serialization occurs. That is, for `downcall_route`, your service will pass in a *Message*, and it will be serialized into a *string*, the type expected by *RouteServiceClass*. For the upcall `deliver`, the *RouteServiceClass* will pass in a *string*, which will be automatically deserialized into a *Message* for your service.

Note that the `method_remappings` block also allows you to specify optional per-message defaults. These appear as declarations of methods with the specific messages given, declared with the appropriate default values. One convenient use for these optional defaults is to set the default service registration id for a message. The syntax is illustrated below. Say that the Ping service used two *RouteServiceClass*es, `r1` and `r2`. If we wanted *Ping* messages to go over `r1` and *PingReply* messages to go over `r2`, we could define that as:

```
method_remappings {
  uses {
    downcall_route(const MaceKey&, const Message& −> const std::string&, registration_uid_t regId);
    downcall_route(const MaceKey&, const Ping&, registration_uid_t regId = r1);
    downcall_route(const MaceKey&, const PingReply&, registration_uid_t regId = r2);
  } // uses
  \ldots
} // method_remappings
```

In this example, we listed the registration id parameter on the remapped method as well, since we will need it on the target methods.

However, when there is only one suitable service variable on which a downcall could be made, then, provided the default registration id is passed (that is, you omit the final parameter in the downcall), then the call will go to the that service. Thus, we do not need to specify any optional parameters for our messages, because we use only a single *RouteServiceClass*.

H RPC

TBD