Contents

1 Introduction .................................................. 3

2 Overview of Modules ......................................... 3

3 InnerRM ......................................................... 4
   3.1 InnerRM Subclasses ..................................... 5
      3.1.1 SimpleComputeRM ................................. 5

4 Transaction Manager ........................................ 6

5 Booking Manager ............................................. 6
   5.1 Booking .................................................. 7

6 Timetable ....................................................... 8

7 Simple Booking Manager .................................... 8

8 Assistant Classes ........................................... 9
   8.1 AcceptorDoc ............................................. 9
1 Introduction

This document describes the proposed modules for the HARC RMs.

Notationally, in this document, information about errors is given in italics.

2 Overview of Modules

A simple block diagram of the RM is shown in Figure 1. The modules are as follows:

**OuterRM** This module calls MainLoop::initialize to set up the other modules, then sets up the HTTP listener.

**MainLoop** This handles requests and passes them to the InnerRM, except in the case of Booking requests, which get routed to the TransactionManager. In the initialize call, MainLoop will create an object of type InnerRM, based on environment variable settings. Currently, there is only a SimpleComputeRM subclass implemented.

**TransactionManager** This module looks after all the ongoing transactions that the RM knows about. A transaction only covers the construction or cancelation of bookings—they don’t last any longer than the commit/abort of the booking. The TransactionManager calls into the InnerRM code.

**InnerRM** All RMs can be implemented as a new subclass of InnerRM. There is some default code in this class to handle a set of Actions. Some implementations may override this code, which is fine.
**BookingManager** Looks after tentative and confirmed bookings.

**Timetable** Support class for keeping track of the booked/free resources.

**AcceptorDoc** Provides access to the contents of documents that arrive from the Acceptors. Also has a simple method to construct a Phase2A message which is sent back to the Acceptors in response to a Prepare message.

**ResourceChunk/Interval** Simple classes to represent chunks of booked resource in a timetable.

**Util** A set of simple utility functions, mostly to assist in parsing XML messages.

The design goals for the refactoring of the code were as follows:

1. All aspects of the Paxos Protocol should be hidden from the InnerRM, including the re-fetching of the transaction status from the Acceptors if the Commit/Abort message is not delivered. The InnerRM also doesn’t need to worry about duplicated messages.

2. Basic tools are provided for the creation and maintenance of timetables and for managing bookings of subresources within the RM. The tools are flexible and extensible, but also optional.

3. All coding effort for new RMs is kept to the InnerRM code.

4. The use of the support modules (everything below the InnerRM in the diagram) is entirely optional.

5. If the provided BookingManager is used as documented, then the InnerRM code won’t have to worry about persistence—this will be handled by the TransactionManager and BookingManager modules.

### 3 InnerRM

As stated above, the hope is that the InnerRM is at the right level to support any new type of RM. The default code in the InnerRM supports subclasses like SimpleComputeRM which handle each action as an individual request, even if they’re part of the same transaction. Clearly that’s not going to be sufficient for the GLambda HGLW, and they may end up overriding nearly all of the InnerRM methods, and/or replacing the module entirely. But we’ll see.

The benefit of going with the simple scheme is that you’ll get persistence for free (when I implement it). But for the HGLW, that’s probably not such a big deal, as you are really wrapping another system which is capable of maintaining the state.

I want to add a fuller description of the InnerRM here.

---

1There is a slight caveat. The InnerRM is passed an AcceptorDoc object, rather than the raw XML. However, the XML is easy to get to, and the API of AcceptorDoc does not have to be extensively used.
3.1 InnerRM Subclasses

3.1.1 SimpleComputeRM

Models a single compute resource, which consists of a number of identical CPUs. Memory can be configured in one of three ways:

1. Memory is a bookable resource, independent of the CPUs (shared memory model);
2. Memory is available at a fixed amount per CPU (distributed memory model);
3. Memory requirements are ignored completely.

Memory requirements are checked if they are specified. In the case where the memory is a reservable independent resource, specification of required memory is mandatory.

The SimpleComputeRM accepts Work elements in two formats. The first is a simple, native format, and looks like this:

```xml
<Work>
  <SimpleCompute>
    <CPUCount>64</CPUCount>
    <MemoryMB>61440</MemoryMB>
  </SimpleCompute>
</Work>
```

The second is a JSDL-based format. The same request would look like this:

```xml
<Work>
  <JobDefinition xmlns="http://schemas.ggf.org/jsdl/2005/06/jsdl">
    <JobDescription>
      <Resources>
        <TotalCPUCount><Exact>64</Exact></TotalCPUCount>
        <TotalPhysicalMemory>
          <LowerBoundedRange>64424509440</LowerBoundedRange>
        </TotalPhysicalMemory>
      </Resources>
    </JobDescription>
  </JobDefinition>
</Work>
```

Using the JSDL format has the advantage that if you’re using JSDL anyway, then you don’t have to reformat your XML.

Bookings are kept in a timetable, and this is checked first, rejecting bookings which would overbook the timetable. If this succeeds, then the scheduler is asked to make the booking. It’s decision is final, of course, and may fail even if the SimpleComputeRM thinks it should succeed.
4 Transaction Manager

This module keeps track of all the transactions that the RM knows about.
Each transaction is identified by a unique string. This consists of:

• The name of the acceptor set (globally unique);
• The TID.

And they can be looked up by specifying this. Although that’s not done by the customizable RM part. This identifier is passed in instead.

Things that are kept here:

1. The endpoints of the Acceptors for the transaction;
2. The time that the transaction started.

The API will be:

handleMessage($dom_doc);

Passes the TM a new prepare/commit/abort message. This will invoke methods in the RM Implementation. It deals with outstanding transactions, and so on. The Inner RM API doesn’t have to cope with details of sending/receiving messages, or worry about receiving duplicate messages. This will all be handled by the Transaction Manager.

There is no API exposed to the Inner RM.

5 Booking Manager

A booking is an entry in a schedule. It consists of:

• A mapping from resource names to ResourceChunks. The resource name must correspond to a timetable that the RM knows about.

A booking can be looked up in a couple of ways:

• By transaction (you get a list).
• By the resource name, and the ident of the reservation. This is the only mechanism that should be needed by the RM.

This module is the one that the RM will go through to make bookings. By doing things this way, I can look after all the persistence stuff.

The API will be:
newMakeBooking creates a new, empty Booking, which is registered to the transaction identified by $transHandle and is for the named resource—this might be something like "mike4.cct.lsu.edu". You also specify the start and end time when you create the new booking, as this is a booking of a single resource—even though it might consist of subresources, such as paths in a network, or CPU and Memory, these are booked for the same time. addCancelBooking adds a Booking to the transaction, which must have already been through a successful tryCancel (to be TENTATIVELY_CANCELED).

Bookings are fetched either as a list for the transaction, or using getBooking. completeTransactionBookings can be used to abort/commit the pending actions on the Bookings.

Also:

    my $ident=getNewBookingIdent();

This can be used by things like the NRM, which doesn’t have an underlying scheduler that handles bookings.

See also the Booking Object API in the following section.

5.1 Booking

Most of the RM’s interactions with bookings are done through these object.

The booking has an internal state, which records it’s status:

UNREGISTERED A new booking (after getNewBooking) not yet registered, during construction. Unregistered bookings are not persistent.

REGISTERED A booking which has entries in timetables. Bookings in this state have the following substatus:

    TENTATIVELY_MADE This is the initial state for a registered booking. It has yet to be confirmed. If the Make is committed, then this will go to the CONFIRMED state.
    CONFIRMED The booking has been committed. There are no outstanding actions.
    TENTATIVELY_CANCELED There is a pending Cancel action on the booking.

Other states, TENTATIVELY_MOVED and TENTATIVELY_MODIFIED will be added. Once committed, these will go to CONFIRMED. Once aborted, these will also go to CONFIRMED (unchanged).
There's probably no need to keep canceled bookings around. I'll delete bookings once they become canceled.

$ok=booking->trySubResourceBooking(subresource,[amount]);

This tries to add a subresource booking, identified by the subresource parameter. Amount is specified if and only if the subresource is quantitative.

$ident=booking->getIdent();

$booking->setIdent($ident);
$booking->registered($resource_name,$ident);

getIdent returns the ident if set, undef otherwise.
registered moves the booking to the confirmed state for the first time (substate TENTATIVELY_MADE). But this is now called by the BookingManager, where the user goes to register. The user must have already invoked setIdent.

($ok,$error)=$booking->tryCancel();

Tries to cancel the booking by removing the entries from the timetable. This should succeed.

$booking->abortAction();

Aborts whatever action was outstanding. If it's a Make, then the booking will disappear. If it's a cancel, then the booking will remain as it was.

$booking->commitAction();

Commits whatever action was outstanding. If it's a Make, then the booking will become CONFIRMED. If it's a cancel, then the booking will disappear.

6 Timetable

NDY

7 Simple Booking Manager

Provides roughly the same API as the above, but without any knowledge of timetables or intervals or resource chunks. This can be used in place of the BookingManager in the case where all the other information is kept in the underlying scheduler (which is the preferred situation).
8 Assistant Classes

8.1 AcceptorDoc

The InnerRM gets passed one of these from the TransactionManager, rather than the DOM tree itself. This object provides easy access to the parts of the Prepare/Commit/Abort message (the Inner RM only ever gets the Prepare document passed to it).

The API is as follows:

```perl
my $parsed_ok=$prepare->parsed();
my $parse_error=$prepare->parseError();

parsed returns 1 if the document was parse ok, or 0 if it wasn't. (The Inner RM code won’t have to do this.) parseError returns the error if parsing was unsuccessful.

my @acceptors=@{$prepareDoc->acceptors()};
my($url,$tid,$actionsREF)=$prepareDoc->instance();
my @actions=@{$prepareDoc->actions()};

acceptors returns a reference to a list of the Acceptor endpoints. Each is a URL, passed as a string. instance gives you access to the elements which currently define the instance of Paxos which we’re participating in, namely the URL of this RM, the TID of the transaction, and then the list of ActionId elements (this is soon-to-be deprecated). actions passes back a reference to the list of actionIds found in the Action elements themselves.

my $actionType=$prepareDoc->actionType($act);
my $actionElem=$prepareDoc->actionElem($act);
my $a2=$prepareDoc->actionSubElem($act,ACTION_RESOURCE_ELEMENT);

These calls all take an actionId (from the list returned by actions) as a parameter. actionType returns a constant from action_enum, which will be either ACTION_MAKE or ACTION_CANCEL (currently). actionElem returns the DOM Element of the action, either the Make or Cancel element. actionSubElem returns the named SubElement of the action. It doesn’t search the tree—only certain elements are indexed this way. The values that will work depend on the type of action. For any action, ACTION_RESOURCE_ELEMENT will return the Resource Element. For Make, RM_SCHEDULE and RM_WORK can be used to return the Schedule and Work elements. For Cancel, RM_RESV_IDENT will return the Ident element.