Robust Services Core

Software Design

Release v0.34.0

July 21, 2022

CONTENTS

DESIGN NOTES

This section discusses various areas of the design, particularly ones in which significant evolution is planned. When some or all of the evolution has occurred, it usually describes what has been implemented.

OBJECT POOLS

To insulate applications from one another, it can be necessary to create multiple object pools for the same type of underlying object.

1. A separate pool of *IpBuffers* (*BtIpBufferPool*) was created for the *BuffTracer* tool. This was done because, when *BuffTracer* was the only tool running under load, it quickly held so many cloned *SbIpBuffers* in the trace buffer that they ran out.
2. *PotsTrafficThread* services all calls queued on its current timewheel entry before pausing. Under heavy load, this has caused messages to run out, because there is no protection against it until the POTS call runs and overload controls discard messages.[1] Although traffic generation should eventually run on a separate processor, an interim solution would be to have separate object pools for the POTS shelf (traffic) and POTS call.
3. When maintenance capabilities are added to support a distributed system, a separate pool of messages will be need to insulate maintenance from payload applications.

LOGS AND ALARMS

This section describes how logs and alarms are now implemented.

1. Each log type registers with the log framework. Each log belongs to a log group and has documentation that can be viewed using the CLI.
2. The first line of each log uses a common format.
3. Each log is placed, as a C string, in the active *LogBuffer*. If the buffer overflows, older logs are retained, given that they are more likely to reveal the cause of the log flood. *LogThread* bundles logs in the buffer and sends a message to *FileThread* to have them written to the active log file. The logs remain in the buffer until *FileThread* acknowledges that they have been successfully written out.
4. A new *LogBuffer* is allocated during each restart. If all logs in the previous buffer have been successfully written to its log file, the buffer is freed during the restart. If some logs remained unwritten because of the restart, the buffer is saved. The CLI can then be used to write the logs to the buffer's log file and free the buffer.

---

[1] When an object pool runs out of available blocks, an additional set of blocks is now allocated immediately, which prevents the problem described here. However, this behavior should eventually be modified to limit the number of pool expansions so that a misbehaving application cannot cause an inordinate amount of memory to be allocated.

5. Logs are not prioritized, but CLI commands support log throttling. All logs in the same group can be suppressed. An individual log can also be suppressed; it can also be throttled, in which case only every *n*th occurrence is saved.
6. The statistics for each log include the number of times it was buffered, suppressed or throttled, and discarded (because the log buffer was full).
7. A log can raise an alarm, which persists until another log clears the alarm. The CLI `>status` command displays active alarms. An alarm can also be cleared manually from the CLI.
8. Each alarm registers with the alarm framework, and each has documentation that can be viewed using the CLI.

OVERLOAD CONTROLS

DETECTING OVERLOAD

The following mechanisms for detecting overload are currently supported:

1. The length of the ingress queue reaching a limit. Because the ingress queue is serviced last, this implies that the system is busy. Accepting more ingress work will probably be futile because the requests are likely to time out before they are serviced.
2. The number of available message objects dropping below a threshold. Cutting off ingress work at this point is important because it guarantees messages for progress work, even if there is no actual overload and the objects were simply under-provisioned.[2]
3. When the delay on the ingress queue reaches a limit. This was implemented as part of queueing ingress messages in FIFO order. The reason for FIFO is that, when the system is heavily loaded, LIFO results in a significant delay (poor service) for *all* ingress requests, whereas FIFO results in good service for at least *some* ingress requests.

These mechanisms allow the POTS application to maintain throughput under heavy overload. CPU idle time was also considered as a sign of overload but was rejected because it could also be caused by work in non-payload factions.

Message loss in I/O threads (*IpBuffer* exhaustion) may occur during overload. In the POTS application, this can be detected by comparing statistics for the number of messages sent by the POTS shelf factory and received by the POTS call factory, or vice versa.

HANDLING OVERLOAD

The following are currently implemented:

[2] The size of an object pool is now automatically expanded when it has no available blocks, so this overload control would rarely be satisfied. However, this control is being retained because an upper limit on the number of automatic pool expansions should eventually be imposed.

1. Discarding an ingress message when the ingress queue exceeds a limit or when the number of available message objects drops below a threshold. See *InvokerPool.RejectIngressWork*.
2. Discarding a retransmitted message. See *Factory.ScreenIcMsgs*.
3. Discarding a context, on which a request message is queued, when a cancellation message arrives. See *Factory.ScreenIcMsgs*.
4. Placing a context created for an ingress message at the head of the ingress queue. This favors new work over old, which conflicts with a "dial tone at all costs" strategy.[3]

The following are possible enhancements:

5. Discarding the oldest context on the ingress queue in order to accept a new context. This also favors new work over old, which again conflicts with a "dial tone at all costs" strategy.
6. Discarding a context that was just dequeued from the ingress queue if its request has probably timed out because it was queued for too long. This would be useful for protocols that lack cancellation messages.

It used to be that the initial PSM and SSM were allocated at I/O level when an initial message arrived. This was changed when refactoring *ProtocolSM* to support protocol stacks. Creation of the PSM and SSM is now deferred until the context is dequeued, which reduces the overhead incurred before discarding a context during a request-cancellation sequence. These sequences start to appear during overload, as new work suffers queue delays and users give up.

ALARMS

A number of alarms have now been implemented. See the `>alarms` command.

BABBLING IDIOT CONTROLS

The POTS application should incorporate babbling idiot controls to demonstrate their implementation. Here is a sketch of the implementation:

1. Add *LeakyBucketCounters* to *PotsCircuit*. Two or three are required:
    a. to detect babbling (perhaps 10 messages in 5 seconds);
    b. to ignore messages until the circuit has built up enough credits for remaining silent;
    c. to detect excessive babbling (perhaps 4 incidents in 30 minutes).
2. Modify the POTS traffic thread to occasionally cause babbling.
3. When babbling occurs, put the circuit in lockout. Once it remains silent, send it a Reset message.
4. If excessive babbling occurs, put the circuit in a new "trouble" state. In this state, the circuit is ignored and cannot receive calls until a CLI command is used to return it to service.

[3] "Dial tone at all costs" means that a user who waits patiently for dial tone (instead of constantly retrying) usually receives it. Because overload controls can discard an initial ingress message, the strategy must also include periodically retransmitting the user's initial message, subject to a backoff strategy that eventually gives up during severe overload.

ESCALATING RESTARTS

OVERVIEW

RSC defines the following types of restarts:

- A **reboot** means restarting the entire program (executable). All data is lost and must be recreated.
- **Reload**, **cold**, and **warm** restarts have less of an impact, as described below.

Restart escalation is from warm (the least severe), to cold, to reload, and finally to reboot. Escalation occurs if the current level of restart fails to return the system to service or if it quickly runs into trouble again.

Escalating restarts are supported by memory types with different persistence and protection characteristics:

- Temporary memory (***MemTemporary***, ***Temporary***) is freed during all restarts.
- Dynamic memory (***MemDynamic***, ***Dynamic***) is freed during cold and reload restarts.
- Slab memory (***MemSlab***, ***Pooled***) is also freed during cold and reload restarts.
- Persistent memory (***MemPersistent***, ***Persistent***) is freed during reload restarts.
- Protected memory (***MemProtected***, ***Protected***) is freed during reload restarts and is write-protected once in service.
- Permanent memory (***MemPermanent***, ***Permanent***) survives all restarts.
- Immutable memory (***MemImmutable***, ***Immutable***) survives all restarts and is write-protected once in service.

When a heap is freed during a restart, *destructors are not invoked*. This is what significantly speeds up recovery time. If a destructor needs to be invoked, a *Shutdown* function (described below) must take care of it.

Escalating restarts also affect threads:

- *RootThread* survives all restarts because it acts as a watchdog to ensure that the restart succeeds.
- *InitThread* usually survives restarts. If it exits, *RootThread* recreates it and the restart escalates.
- *CinThread* survives all restarts because C++ does not provide a way to unblock a console read operation. To overcome this limitation, *CinThread* would have to be implemented with low level keyboard functions, which are platform specific.
- I/O threads survive warm restarts, which attempt to preserve subscriber work and seamlessly resume once the system returns to service.
- All other threads exit during a restart. Excessive trapping is often the cause of a restart, so the strategy is to exit and recreate most threads. The overhead and impact of this are low, and it fixes a sanity problem caused by corrupt thread data.

A separate heap supports each type of memory. The default process heap (the one used by *::operator new* and *malloc*) is also used for classes derived from *Permanent*.

HOW RESTARTS AFFECT DESIGNS

Because they define different types of memory, escalating restarts affect the design of software components. This section summarizes what to consider.

1. Derive each class from the appropriate *Object* subclass: *Temporary*, *Dynamic*, *Persistent*, *Protected*, *Permanent*, or *Immutable*. Classes derived directly from *Object* cannot be allocated on the heap, so they must either be stack objects (e.g., *SysTime*) or members of other objects (e.g., *LeakyBucketCounter*).
2. *Object* defines *Shutdown* and *Startup* functions to support restarts. *Shutdown* functions are analogous to destructors in that they tear the system down to a known state, and *Startup* functions are analogous to constructors in that they build it back up to an in-service state.
3. Be especially wary of causing an exception during a restart. This escalates the restart to the next level, which eventually leads to a reboot. An exception during a reboot leads to a restart loop, which is probably the most embarrassing outcome imaginable for a supposedly robust product.
4. Use *Singleton.Extant* to avoid recreating a singleton prematurely, either during a reboot or a restart.
5. When the system initializes, it does not use write-protection. Write-protection is only enabled when the system enters service.
6. During a restart, *MemImmutable* remains write-protected. During a reload restart, which destroys *Protected* objects, *ProtectedHeap* is not write-protected until the system returns to service.
7. When the system is in service, write-protected data in classes derived from *Protected* can only be changed after defining the stack variable

   ```
   FunctionGuard  guard(Guard_MemUnprotect);
   ```

   This variable unprotects memory and reprotects it when it goes out of scope. These operations are expensive, so the data in classes derived from *Protected* must change infrequently.
8. Write protection may only be appropriate for a subset of a class's data. Split such a class in two using the PIMPL idiom. The primary class derives from *Protected*, and it allocates a *Persistent* struct to hold the data that changes more often. The other pairing is an *Immutable* with a *Permanent*. There are a number of examples of this PIMPL idiom; each of the unprotected structs is referenced through a member with the name *dyn_*.
9. Care is required when a class contains members of lesser or greater persistence. Such members should always be held through a *unique_ptr*.
10. Try to use the function *Restart.Release* and *Restart.ClearsMemory* instead of making hard-coded assumptions about the types of restarts during which an object will disappear.
11. When a class contains a pointer to an object of *lesser* persistence, its *Shutdown* function must clear the pointer during a restart that only frees the object of lesser persistence, and its *Startup* function must reallocate the member. See, for example, the *stats_* member in classes derived from *Protected*.
12. If a class contains a pointer to an object of *greater* persistence, it must be able to *recreate* that pointer. For example, some subclasses of *Dynamic* have members derived from *CfgParm*, which is derived from *Protected*. When one of these classes is constructed, it uses *CfgParmRegistry.FindParm* to look for its configuration parameter, and only instantiates it if it does not yet exist.
13. A class derived from *CfgParm* may need to override *RestartRequired*.
14. A *static const* member or a *const* object defined at file scope (such as a *const string*) behaves as if it were derived from *Immutable*, which is unlikely to cause any difficulties.
15. A *static* non-*const* member behaves as if it were derived from *Permanent*. If the class that defines such a member does not survive a restart, it typically needs to reinitialize the static member during the restart.

See, for example, *BcSsm.ResetStateCounts*. Another example is the *Singletons* registry, which was added so that each affected singleton would not have to clear its *Instance_* pointer during a restart. In some cases, the need to reinitialize can be avoided by making the member non-static, perhaps by moving it to a singleton.

16. A *unique_ptr* member
    a. should invoke *release* if the *unique_ptr* will survive the restart but the underlying object will not (to avoid the overhead of invoking the object's destructor): see *Restart.Release*
    b. should invoke *reset* if the *unique_ptr* will not survive the restart but the underlying object will, unless the object can be found again after the restart (to avoid leaking the object)
17. An object allocated on the default heap survives all restarts and must therefore
    a. be owned by another object that survives all restarts, or
    b. be explicitly deleted by a *Shutdown* function.

    To avoid a memory leak in (b), the object may have to be allocated using *new* (that is, it may need to be a member pointer). For example, a C++ string cannot be forced to free its memory (*string.resize* truncates the string without freeing memory, and *string.shrink_to_fit* is not guarantee to free memory). The string must therefore be deleted outright to avoid a memory leak.
18. With regard to the previous point, vector heap allocation (*new[]* and *delete[]*) often needs to be avoided. POD arrays (including arrays of pointers) are allocated on the default heap and will therefore leak if owned by an object whose heap is freed. To avoid the cost of explicit deletion in the enclosing object's *Shutdown* function, and to place the arrays in the desired type of memory, use *Memory.Alloc* to allocate them and *Memory.Free* to free them.
19. Call *Object.Cleanup* where necessary. For example, during cold and reload restarts, *Thread.Shutdown* invokes *Cleanup* on all *MsgBuffers* queued on threads. The reason is that *MsgBuffer* uses *MemDynamic*, whose *StreamRequest* subclass owns an *ostringstream* that must be deleted to avoid a memory leak. The *delete* operator could be used instead, but *Cleanup* is faster.
20. A derived class can override the type of memory used by its base class by overriding *operator new*. However, this can be problematic because it is apt to make the base class's *Shutdown* and *Startup* functions unusable.
21. By default, an STL container allocates memory from the default heap. If such a container is a member of an object that does not derive from *Permanent*, its dynamically allocated data will be leaked during a restart, when the heap for the object that owns the container is deleted without invoking the object's destructor. The appropriate allocator in *Allocators.h* must be assigned to the STL container to overcome this problem. *NbTypes.h* defines various strings, such as *DynamicStr* and *ProtectedStr*, for this purpose.

ENHANCEMENTS

At present, RSC supports warm and cold restarts. The following capabilities still need to be implemented:

1. **Reboot**. This is supported by *RscLauncher*, which forks RSC as a child process after prompting for the path to its executable. If a reboot is required, RSC exits with a failure code and *RscLauncher* relaunches it.
2. **Reload restart**. This has been implemented.
3. **Escalation**. If the system gets into trouble soon after a restart, the restart level should escalate. This is currently done if an exception occurs *during* the restart, but it should expand to include an interval after the restart ends. When the *LeakyBucketCounter* for "too many traps" is implemented, the time when the

restart ended can also be saved so that the restart can escalate if too many traps occur during the first five minutes of service.

4. **Journal file**. After a reload restart, *MemProtected* needs to be rebuilt. It consists mostly of configuration data, such as subscriber profiles. A journal file logs commands that populate configuration data and reruns them so that the same configuration exists after the reload restart. The journal file cannot be run until the restart ends, and other threads should not run while the journal file is being processed. If a faulty provisioning command caused a restart, it should not be reapplied if it was entered in the journal file. It is desirable, once a certain amount of provisioning has occurred, to clean up the journal file. This can be done by prompting provisioned data to output the commands that would recreate it. Running these commands is a slower alternative to restoring provisioned data from a binary database (see below).
5. **Write protection**. This has been implemented.
6. **Binary database**. A relocatable binary image of *MemProtected* should be supported. This image is loaded during a reload restart, after which the journal file only has to rerun the provisioning commands that occurred *after the image was saved*. In a large system, this significantly reduces the downtime for a reload restart.
7. **Module initialization order**. Each *Module* constructor instantiates the modules that it requires. This has the effect of adding modules to *ModuleRegistry* in a correct initialization order (a partial ordering).

REJECTED APPROACHES

It is often useful to discuss ideas that were considered but not included in a design. For escalating restarts, the following were rejected:

1. Using *PegCount* to track memory allocations (successes and failures) and deallocations in each *SysHeap*. This was rejected because a *PegCount* is reset at the start of each 15-minute statistics interval and is lost entirely during cold and reload restarts. Because the purpose of these counts is to detect memory leaks, it is undesirable for them to be reset or lost.
2. Using the same design as each *Module* subclass (a *Registered* bool initialized by a *Register* function) to bind per-class static *Shutdown* and *Startup* functions. This would provide a finer level of granularity than appears to be unnecessary, at least so far.
3. Adding a new module below *NbModule* to handle, during a restart, the classes that *RootThread* handles during a reboot. Or alternatively, always exiting *InitThread* so that *RootThread* can handle these classes. Instead, *NbModule* simply handles the few classes in question.
4. Validating the heap(s) that will survive a restart, and escalating the restart if a heap that was supposed to survive is corrupt. Because of the overhead of heap validation, this was rejected in favor of allowing the restart to escalate if a corrupted heap causes an exception.
5. Zeroing memory that will not survive a restart rather than freeing it. Placement *new* would then be used to recreate objects in this memory. This was rejected because a simple design for it is elusive. The savings from not freeing and reallocating memory are unlikely to offset the additional complexity.
6. Overriding global *operator new* and *operator delete*. This is a major undertaking because even class templates in the standard library would use the overrides. A proper implementation calls for at least eight functions: *new*/*delete* x scalar/vector x *throw*/*nothrow*. This becomes sixteen if global *operator new* also supports a *MemType* parameter, which would be the point of the whole exercise. There is also the issue of properly supporting *new_handler*.

MEMORY ALLOCATION FAILURES

Given that *Object* and its subclasses had to override *operator new* to support the various types of memory required for escalating restarts, the question of how to handle memory exhaustion arose.

Standard practice is to throw *bad_alloc*. The use of this exception, and all others defined by C++, is undesirable because it does not capture a stack trace. A stack trace may not be particularly useful when memory allocation fails, but capturing a trace is the general principle. The goal is to always throw a subclass of *NodeBase::Exception*, which captures the stack. *AllocationException*, for example, replaces *bad_alloc* as follows:

1. RSC memory allocation functions, such as *ObjectPool.DeqBlock*, throw *AllocationException* directly.
2. When invoking global *operator new*, RSC functions should specify the *nothrow* version. If it returns *nullptr*, the RSC function can then throw *AllocationException*.
3. If global *operator new* does not support a *nothrow* version, a RSC function that invokes it must catch *bad_alloc* and throw *AllocationException* in its place.
4. Given that RSC uses exceptions to recover from extreme errors, throwing *AllocationException* is far better than returning *nullptr*. The latter is only appropriate if it avoids an even bigger problem, such as throwing an exception when memory is being allocated from
    a. An interrupt service routine. Objects allocated at interrupt level should override *operator new* with a version that invokes a static *atIoLevel* function. If this function returns *true*, *nullptr* would be returned instead of throwing *AllocationException*. The relevant *InvokerPool* functions, *ReceiveBuff* and *ReceiveMsg*, already have an *atIoLevel* argument.
    b. A destructor. This is usually dubious, but the *ProtocolSM* destructor invokes *SendFinalMsg* if the PSM is not in the *Idle* state, and this function needs to allocate a message. The *atIoLevel* function mentioned in (a) should therefore be extended to support this scenario.

DISTRIBUTION

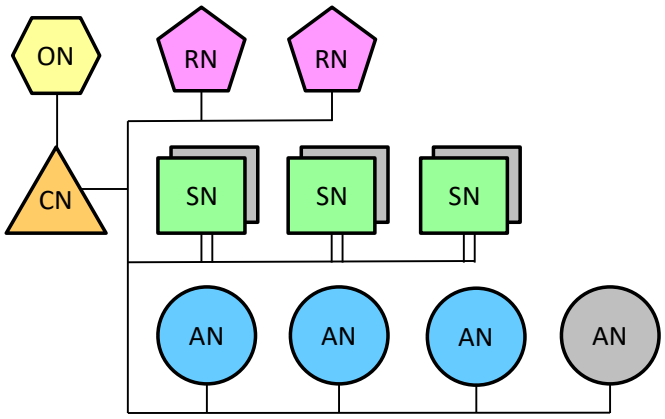
The POTS application that is constructed using RSC currently combines, in a single executable, the functions of the *administration* (referred to as *operations* in this document), *control*, *service*, and *access* nodes described in *Robust Communications Software*. The primary way that RSC needs to evolve lies in supporting robust, distributed networks rather than simply robust, independent nodes. This section outlines the necessary enhancements.

TERMINOLOGY

- A **processor** runs zero or more executables.
- A **node** has one or two processors. If two, the purpose is to support hot or warm standby, or load sharing.
- An **element** has one or more processors. If more than one, it supports symmetric multiprocessing.
- A **role** is a node's function within a network, such as an *operations*, *control*, *service*, or *access* node.
- An **executable** fills one or more roles. If more than one, the executable is a "combo load".

NETWORKING

Heterogeneous Distribution. The POTS application is currently one big node that combines the functions of the operations, service, and access nodes. Because it is a standalone application, it lacks control node functions. To provide a testbed for distribution, its single executable will be split into the following nodes:



ON Operations Node
CN Control Node
RN Routing Node (load sharing)
SN Service Node (warm standby)
AN Access Node (cold standby)

- A combined, standalone control/operations node. Operations nodes are not core network elements and need not be carrier grade, so there is no plan to develop frameworks for such nodes. Integrating the operations node and control node will reduce the amount of effort required. Control node functions, however, are a primary area where new capabilities need to be developed.
- Two or three service nodes, each with a warm standby, for the POTS call.
- Two or three access nodes, sharing a cold standby, for the POTS shelf. Each access node will also be able to serve as a *traffic node*, which generates POTS calls during load testing.
- Two **routing nodes** in a load sharing configuration, for finding the terminating service node in a POTS call. Each contains a database that maps a directory number to its associated service node. The POTS call will send a message to this node before it enters the Selecting Route state. A modifier SSM will implement the protocol, because all originating basic calls need to send this message.

Access Node. Some of the changes required to split the POTS application into separate service and access nodes include

- Split *PotsCircuit* and partially replicate it between the service and access nodes.
- Synchronize port numbers and DNs by downloading provisioned data to service nodes and access nodes during commands such as >register and >deregister.

- Move *PotsTrafficThread* to the access node. A mechanism for creating traffic DNs will be needed.
- Find a way to support the >inject command now that *PotsShelfFactory* is not on the service node.

So far, the discussion has focused on the changes required to distribute the POTS application. The remainder of this section discusses distribution topics from a more general standpoint.

Loading software. For now, each node's executables are assumed to reside on a local disk, where they are placed using a service provided by the underlying platform (e.g. FTP).

Element discovery. One of a node's executables is the loader mentioned in "Enhancements" on page 9. It is launched using the underlying platform and is deliberately simple: it launches a **local maintenance** executable, monitors its status, and shuts it down and restarts it if it fails to respond. The element's configuration file contains the operation node's address, allowing local maintenance to register with the operations node. The registration message includes the element's attributes, its IP address, and the roles that it can support. Local maintenance will obtain the element's attributes (such as processor type, number of processors, and memory capacity) from a new *SysElement* object, and it will create the list of roles from configuration parameters (tuples that map a role to the executable that supports it). As with other *Sys** objects, SysElement.h will be generic, with platform-specific .cpps.

Configuring nodes. Registration messages allow the operations node to assemble a list of elements and their capabilities. The first role to establish is a control node. This is done by selecting an element for that role and having the operations node send a message to that node's local maintenance function to enable the role. Local maintenance then launches the executable for the control node role, and that executable also registers with the operations node. Messages to configure other nodes go through this control node so that it can assemble a view of the nodes for which it is responsible. On each of these nodes, local maintenance acts as the control node's agent and manages the other executables. Local maintenance can terminate an executable and relaunch it, either independently or at the behest of the control node. It can also launch a new executable when told to do so by the control node. In a multiprocessor system, it distinguishes instances of the same executable. A processor number can be used for this purpose, as each processor will be restricted to running one instance of any given executable.

When a node is provisioned, it needs to be given IP port numbers for the services that it will provide. Similarly, it needs to be given IP addresses and port numbers for services that it will obtain from other nodes. The information required depends on the node's role.

Symmetric multiprocessing. One question is how to handle a node that supports SMP. By default, such a node has one IP address. RSC will use the approach outlined in *Robust Communications Software*: treat processors in the SMP cluster as separate nodes, and send messages between them. This raises the question of whether to assign a separate IP address to each processor or use the single IP address assigned to it.

Assigning separate IP addresses allows each processor to use well-known port numbers (e.g. port 5080 for SIP). This is a significant advantage. With a shared IP address, each well-known port would become a *range* of ports (with the processor number in the four low-order bits, for example). Regardless of the approach, an executable must be provided with its processor affinity.

Proprietary shelves. In a shelf, the insertion or removal of a card raises an event. To support a proprietary shelf, the control node would have to incorporate these events. A "card inserted" event is similar to registering with the

control node, as previously discussed in this section. The inserted card would presumably be equipped with a proprietary loader so that software could be downloaded to it.

Redundancy. The control node should support the redundancy strategies of load sharing, cold standby, and warm standby. These strategies must be specified at configuration time so that the control node can react accordingly when the system is in service and a failure occurs. The control node must monitor the state of all nodes and inform affected nodes when a collaborating node changes state. These nodes can then initiate their own recovery actions (e.g. by invoking *InjectFinalMsg* on affected PSMs).

Operations. The control node should support the following commands, which are coordinated with the local maintenance software running on the affected node:

- *Query* the status of the node.
- *Load* software into the node.
- *Enable* the node (i.e. place it in service).
- *Disable* the node (i.e. remove it from service).
- *Test* the node if it supports processor and memory diagnostics.
- *Reset* the node (i.e. force a restart).
- *Failover* to the mate node in a warm standby configuration.
- *Synch* with the mate node in a warm standby configuration.
- *Abort* the operation currently in progress.

NETWORKED THREADS

During normal operation in a distributed system, only the operations node has a “console”. In fact, it is likely to be graphical and multi-user. A control, service, or access node might have a console, but it would only be used to access functions not available from the operations node. This would be rare, because these nodes might well be geographically dispersed. And to reduce costs, these nodes might have neither a console nor a disk.

To support a node without a console or disk, RSC must evolve to support input from, and output to, remote consoles and files. This affects the following threads:

- *CinThread*
- *CoutThread*
- *FileThread*
- *StatisticsThread*
- *LogThread*
- *CliThread*

Remote consoles and files would probably be supported using TCP, in which case *StatisticsThread*, *LogThread*, and *CliThread* would evolve to access them through an *IoThread*. This implies that the local versions of *CinThread*, *CoutThread*, and *FileThread* should evolve to present the same type of interface as an *IoThread*.

For example, one way to evolve *CliThread* would be to refactor it into local and remote versions. The local version would run on the operations node; the remote version, on control, service, and access nodes. Some of the changes would be to

1. Change *CinThread* to send a *MsgBuffer* to its client (one of the behaviors of an *IoThread*).
2. Support the >read and >copy commands on the local CLI thread.
3. Implement a >login command (similar to telnet) on the remote CLI thread.
4. Commands for the remote CLI thread arrive when its *TcpIoThread* invokes *recv* and queues a *MsgBuffer* for the thread.
5. Have the remote CLI thread use *send* to report the results of a command (from *CliThread.obuf*).

DATA DISTRIBUTION

One attribute of a carrier-grade node is that its configuration data can be modified while it is in service. A truly carrier-grade system takes this farther, by automatically propagating configuration data changes to all affected nodes. This requires a distributed data framework (DDF) that tracks which nodes have acknowledged a change. DDF can then re-notify an affected node if a message gets lost or the node was out of service.

RSC needs to provide a DDF capability. This is challenging because there are different types of configuration data, which are modified by different software components. A central question is whether nodes (or node types) register to be notified of changes, or whether a data field knows which nodes (or node types) need to receive it. Either way, there needs to be a uniform way to associate a data field with the nodes that need it, and a uniform way to inform DDF that a field has changed.

CHECKPOINTING AND FAILOVER

To support warm standby, checkpointing and failover need to be implemented. The high-level design is

1. At the end of a transaction, invoke *Object.Serialize* on the objects in a context. Start with the root SSM, because whether to serialize is an application-specific decision. A POTS call server, for example, is unlikely to serialize a call unless it is in the Active state and has no modifier SSMs.
2. Place serialized objects in a message and send it to a checkpointing thread on the standby node.
3. Have the checkpointing thread on the standby node invoke *Class.Deserialize* on each serialized object's class to recreate the objects.
4. Modify escalating restarts so that a warm restart escalates to a failover to the standby node rather than a cold restart.
5. To make a failover transparent to other nodes, assign four IP addresses to an active-standby pair, node1 and node2:
    a. An address that routes to the active node. Most messages are addressed to this node.
    b. An address that routes to the standby node. The operations and control nodes use this address to perform functions that specifically pertain to a standby node.
    c. An address that routes to node1, and an address that routes to node2. Only node1 and node2 use these addresses.

When failover occurs, update routing tables so that the IP addresses for the active and standby node point to the other node.

DEBUGGING TOOLS

OVERHEAD

Individually enabling trace tools while running POTS traffic reveals that the performance impact of these tools is roughly as follows:

- Function tracer: 20%
- Memory tracer: 1%
- Object tracer: 3%
- Message tracer: 20%
- Transaction tracer: 1%
- Context tracer: 2%
- Immediate tracing: unusable unless running traffic at less than 3% of the maximum rate
- Stack checking: 2% (on every 20th function calls) to 4% (on every function call)
- Tracing context switches: 1%

The overhead of function tracing could be considerably reduced by implementing a function that only returned the level of function nesting (the number of stack frames) instead of also constructing information about each frame.

GENERAL TESTING

The function tracing mode that simply counts how many times each function was invoked (*CountsOnly*) can support code coverage. The functions invoked during a testcase would be recorded against that testcase, and the reverse mapping (function to testcases) would be generated later. This would allow suitable testcases to be selected to retest a function whose code was being changed. The function database could be created within the >ct increment, which already analyzes invocations of *Debug::ft*. By populating the database with all traceable functions, it would also be possible to find those not invoked by any testcase, so that testcases for them could be developed. The function database would be periodically updated to account for new and deleted functions.

A testcase database is also required. For this purpose, each testcase requires an identifier. In addition to the functions invoked during the testcase, the time when it was last executed, and whether it passed or failed, would be recorded. This would enable queries for testcases that failed or that had not recently been executed.

SESSION TESTING

The following capabilities would improve the session test environment:

1. *Injecting a timeout message to expedite a timeout*. This requires cancelling the timer and setting the object pointer in the timeout parameter. The pointer could be set from a command that finds an SSM or

PSM. But if the timer already exists, it is easier to simply find it in the timer registry and force it to expire. The PSM receiving the timeout could be identified using a test session identifier, in which case the operation could be modeled as injecting a timeout message from a test PSM.

2. *Discarding a message to test a lost message scenario*. This only applies when a true peer exists, because a test PSM can withhold any message or response. The message can be discarded when it is sent or when it arrives. A *MsgHeader* field could be added to support this.

3. *Synchronizing two messages to test a message crossing scenario*. The first message could be held
    a. at the source: send it when the second message arrives, but before it is processed;
    b. at the destination: receive it after the second message is sent.

    Holding the message at the source is preferable because the second message could idle the PSM or the entire context. A *MsgHeader* field could be added to support this.

TRACE TOOL PROPAGATION

The ability to trace all contexts in a session is desirable. This can be supported by adding trace flags to *MsgHeader*, one per trace tool. When a context is traced, the flag associated with each active trace tool would be set when sending a message. When a context received a message, it would look at these flags and start the trace tools that were not active on that context. This behavior should be optional so that tracing can still be restricted to a specific context.

TRACE TOOL NETWORKING

Trace tools must be networked in a distributed system. For example, it should be possible to enable a specific trace tool in all service nodes and/or all access nodes. When the tool is stopped, all nodes should send their results to the control node.