Robust Services Core

Software Design

Version 1.4

July 24, 2017

CONTENTS

DESIGN NOTES

This section discusses various areas of the design, particularly ones in which significant evolution is planned.

OBJECT POOLS

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. Although traffic generation will eventually run on another processor, a temporary solution would be to use separate object pools for the POTS shelf (traffic) and POTS call. Eventually this capability will be needed anyway, to insulate maintenance and payload applications from one another.

An initial version of this capability has been supported by creating a separate *SbIpBuffer* pool for the *BuffTracer* tool. This was necessary because, when *BuffTracer* was the only tool running under load, it quickly held so many *SbIpBuffers* in the trace buffer that they ran out.

To support more than one pool for a given subclass of *PooledObject*, the pool identifier (*ObjectPool_Id*) for each *PooledObject* was moved out of the object, to a header immediately preceding it. This allowed the *ObjectPool_Id* argument to *PooledObject*'s constructor to be removed. More importantly, it prevents the identifier from being cleared when deleting a *PooledObject*, allowing *operator delete* to return the block to the correct pool when the subclass (*SbIpBuffer* in this case) has more than one pool. Finally, because *PooledObject.operator delete* can now return any block to the correct pool, subclass-specific overrides of *operator delete* were either removed or modified to invoke *PooledObject.operator delete*.

To support its two pools, *SbIpBuffer.operator new* had to take a new *SbPoolUser* argument (an *enum*), which it uses to select the correct pool based on the possible values of *PAYLOAD_USER* and *TOOL_USER*. To insulate maintenance and payload applications, this *enum* would be expanded to include *MTCE_USER*.

LOGS AND ALARMS

The log framework requires a number of enhancements:

1. Each log type should register with the framework. Currently, logs are generated on something of an *ad hoc* basis. An inventory of log types is needed for documentation purposes, and each log should have a priority and belong to a log group.
2. Log headers should use a common template. This is largely the case already, with *Logs.Create* taking a string argument and appending the string returned by *Element.strTimePlace*.
3. Each log is currently placed in a *StreamRequest*. This is a subclass of *MsgBuffer*, which is used to send a message to a thread, in this case *LogThread*. Log floods have consumed all *MsgBuffers*, so a separate pool of log buffers should be defined, or there should be a limit on the number of *MsgBuffers* assigned to logs. Given that logs need to be prioritized, a separate pool is appropriate so that each log can be placed on a work queue associated with its priority. *LogThread* would either wake up frequently or be interrupted to

service its work queues. It would also apply its own form of overload controls. If its work queues were backing up, it would discard low priority logs and stop allocating buffers for them.

4. The framework should sort logs so that, when it displays them, it can suppress duplicate logs and favor logs of higher priority. A *LeakyBucketCounter* for each log type might be appropriate. When a log flood occurs, the impact on the console should be reduced by only displaying some logs and/or only displaying their headers. The full version of each log should still be written to the log file if possible.
5. The framework should incorporate statistics to report the number of logs generated in each group, along with their severity. The report would be a table, with the rows being groups and the columns being priorities.
6. The framework should support alarms. A log can trigger an alarm, which persists until another log indicates that the situation which triggered the alarm no longer exists.

Another possible change is to offload log formatting (using *operator* <<) to *LogThread*. An application would save raw data in the log buffer, and *LogThread* would later invoke a callback to format this data. The purpose would be to move formatting overhead into *LogThread's* faction, but this is not straightforward. Small logs might as well be formatted immediately. The overhead occurs with large logs, but these contain strings—stack traces and object dumps—that are difficult to capture as raw data. And to defer formatting, each log must subclass the log buffer with its own member data, which is tedious. This change is therefore not planned.

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.

These two mechanisms allow the POTS application to maintain throughput under heavy overload. Consequently, other detection mechanisms have not been implemented. They are also less straightforward, so there is no plan to include them:

3. When the delay on the ingress queue reaches a limit. This is redundant because the delay is generally proportional to the queue length. It would also be problematic if the ability to add work to the head of the queue was eventually supported.
4. When the CPU idle time drops below a threshold. This is misleading, however, if a transient burst of work occurs in non-payload factions.

Care is needed when provisioning the number of message objects. When a modest workload (perhaps 20% of the maximum capacity) suddenly turns into overload (perhaps 120% of the maximum workload), message objects can be exhausted even if this does not occur when the workload ramps up more gradually.

Message loss in I/O threads (IP buffer overflows) 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:

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*.

The following are possible enhancements:

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.[1]
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

An alarm should be raised when the system enters overload. Both minor and major alarms should be defined. A minor alarm would persist during the initial phase of overload, which is characterized by discarding retransmitted

[1] "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.

messages or request-cancellation message pairs. A major alarm would persist during more severe overload, which is characterized by the immediate discarding of ingress messages.

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.

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 (***MemTemp***, ***Temporary***) is freed during all restarts.
- Dynamic memory (***MemDyn***, ***Dynamic***) is freed during cold and reload restarts.
- Protected memory (***MemProt***, ***Protected***) is freed during reload restarts and is write-protected once in service.
- Permanent memory (***MemPerm***, ***Permanent***) survives all restarts.
- Immutable memory (***MemImm***, ***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. It must, because it is the thread on which *main* runs.
- *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 permanent memory.

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 (usually *Protected* or *Dynamic*). By deriving from one of these classes, or from *Temporary*, *Permanent*, or *Immutable*, a class avoids the need to override *operator new* to allocate the type of memory that it uses. Classes only derive directly from *Object*
    a. if they are intended to be constructed within other objects (e.g. *LeakyBucketCounter*)
    b. if deriving from one of the *Object* subclasses would cause an infinite loop during construction (e.g. *SysHeap* trying to allocate from its own heap; *Singletons* trying to register with itself)
    c. if their subclasses have different persistence requirements (e.g. *Tool*, which is the base class for *TraceBuffer* (*MemPerm*) but also other tools that use *MemTemp*.
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 is in service, write-protected data (in classes derived from *Protected*) can only be changed after invoking *Thread.MemUnprotect*, which is followed by a call to *Thread.MemProtect* once the changes have been made. These functions are costly, so the data in classes derived from *Protected* must change infrequently.
6. Write protection is often appropriate for only a subset of a class's data. If so, split the class into two: a class derived from *Protected*, and a class derived from *Dynamic*. These two classes will often be friends. (The other pairing is a *Immutable* with a *Permanent*, but applications rarely derive from these classes.)
7. 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*.

8. 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.
9. 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.
10. 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.
11. 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.
12. 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. For an example, see *ObjectPool.AllocBlocks*.
13. 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 *MemDyn*, 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.
14. 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.
15. 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.

ENHANCEMENTS

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

1. **Reboot**. The most generic way to support reboots is with a **loader** that launches an executable and monitors its status. If it fails to respond, the loader terminates it and relaunches it. (Windows defines *RegisterApplicationRestart* for automatically rebooting an executable. However, an executable cannot ask to be rebooted; a reboot only occurs if Windows itself detects that the executable is in trouble.)

2. **Reload restart**. This frees most memory, so it should be easy to implement.
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, *MemProt* 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**. Making *MemFirm* and *MemProt* read-only requires the use of platform-specific functions (*VirtualAlloc* and *VirtualProtect* on Windows).
6. **Binary database**. A relocatable binary image of *MemProt* 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**. This should be calculated instead of being hard-coded.

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 template classes 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.

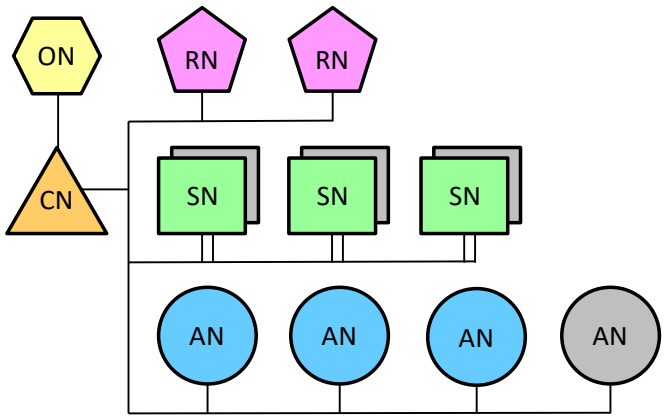
TERMINOLOGY

- 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. Something similar to a table control framework seems necessary.

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.

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

One of the planned function tracing enhancements is to implement a mode that simply counts the number of times that each function was invoked. This would introduce little overhead and would help to guide performance improvements, given that the most frequently invoked functions are the obvious candidates for streamlining.

From a testing standpoint, this enhancement could also support code coverage. The functions invoked during a testcase would be recorded against that testcase, and the reverse mapping (function to testcases) would later be generated. This would allow suitable testcases to be selected when retesting a function whose code was being changed. The function database could be created within the >code increment, which already analyzes invocations of *Debug::ft*. By populating the database with all traceable functions, it would also be possible to detect those not invoked by any testcase, so that appropriate testcases 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 last ran, and whether it passed or failed, would be recorded. This would support queries for testcases that are failing or that have not been executed since a specified time.

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 can 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

In a distributed system, 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.

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.

IMPLEMENTATION NOTES

This section lists work items associated with various areas. In each area, open items are listed first, followed by completed items.

Fonts

black: open item

grey: completed item

Bullets

- ➢ bug
- ○ open item
- ❖ open item requiring major effort
- ? open item that is dubious
- ✓ completed item
- x rejected item
- – second-level bullet

Highlighting

discussed in *Design Notes*

other important enhancement

BASE

- ○ implement *Concatenate* (can't even copy a *Q2Way* to an empty header, because the header itself is included in the *Q2Way* chain)
- ○ support object morphing
- ○ support object templates (cannot use *new*, else constructor chain will be invoked)
- ❖ support auxiliary data blocks (pools, allocation/deallocation, pointer in *PooledObject*): beneficial for patching, else marginal because object pool sizes are engineered, so it is reasonable for a subclass that is larger than its pool's block size to allocate memory from a heap; drawbacks are performance and the risk of memory leaks
- ❖ reboots
- ❖ reload restarts
- ❖ journal file
- ❖ write-protected memory
- ❖ binary database
- ❖ module initialization order

- ✓ refactor *Registry* to create an *Array* template class that provides an expandable array with bounds checking
- x implement option to disable I/O interrupts if queue can be modified at I/O level [no: support for I/O interrupts has been removed, because UDP and TCP are not implemented this way]
- ✓ implement optional mutex for queues
- ✓ provide command to control object pool audit: force execution, change interval (zero = disable)
- ✓ limit nesting of *Debug.SwErr* to prevent infinite recursion
- x support updating node configuration file based on modified values; include key-value pairs for which no configuration parameter is registered [no: journal file will make this unnecessary]
- x modify *Q1Way* so the header points to itself for an empty queue: this would prevent an invalid queue header if the *Q1Link* destructor removes the tail item [no: makes enqueue as expensive as exqueue]
- ✓ object nullification (option to nullify entire block or just *vptr*)
- ✓ add command to display memory in hex
- x add node name to console title: *GetConsoleTitle*, *SetConsoleTitle* [no: Windows specific]
- x use *CreateWindow* [no: control, service, and access nodes might not even have a console]
- ✓ trace initialization starting as early as possible (*Module* binding, work done by *RootThread*)
- x replace *SysStackWalker* with non-managed code [on hold: switching to publicly available *StackWalker* slowed down function tracing by a huge factor]
- x remove compiler /clr option [not possible: needed for *SysStackWalker*]
- x create separate module for *Sys** and its transitive *#include* [no: this is most of *NodeBase* and prevents use of *Debug.ft*]
- ✓ incorrect indentation in function trace: invoker thread stuck at depth=4 [needed to compile with /Z7 so that static library functions weren't essentially opaque]
- x remove **AppIds.h* [no: helps to decouple applications from base]
- ✓ find way to support layered builds: module *Register* functions create the modules that they need, and *main* creates what is wanted in the build
- ✓ create static libraries for *NodeBase*, *SessionBase*, *MediaBase*, and *CallBase*
- ✓ implement test for restart timeout (*InitFlags::DelayInit*)
- ✓ refactor queues and registries into template classes
- ✓ override assignment and copy operators to avoid subclasses of *PooledObject* on the stack
- ✓ bind (unbind) objects in constructors (destructors) whenever possible
- ✓ allow overbinding in bind functions, but generate a log and delete the previous entry if appropriate
- ✓ implement watchdog for initialization timeout in *RootThread*

THREADS

- analyze whether deadlock could occur between RTC lock, context switch lock, and orphan lock
- capturing the stack on an *AllocationException* (created on the default heap) could cause another exception
- add a *LeakyBucketCounter* for the scheduling timeout in *RootThread*
- add a *LeakyBucketCounter* to cause restart if trap threshold is exceeded (*ThreadAdmin.stats_.traps_*)

❖ networked threads (local and remote versions of OAM&P threads)

- kill a thread remotely by changing its instruction pointer (platform specific; on Windows, see *CONTEXT*, *GetThreadContext*, *SetThreadContext*)
- thread heartbeating and re-creation
  - bind a function that is invoked if a heartbeat is not received?
  - *InitThread*, *CinThread*, *CoutThread*, *FileThread*, *LogThread*, and *TimerThread* are recreated when a client indirectly invokes their *Singleton.Instance*
  - a replacement *InvokerThread* could be created by *InvokerPoolRegistry.UnbindThread*
  - a replacement *IoThread* could be created by *IpPort*, if it was notified of the deletion
  - also need to consider *CliThread*, *StatisticsThread*, and *ObjectPoolAudit*

❖ proportional scheduling

  - define timewheel profiles: payload, maintenance, administration
  - create a default timewheel for each profile
  - add a command for editing a timewheel profile (percentage of time allotted to each faction)
  - add a command for manually switching to another timewheel
  - automatically switch to payload timewheel when system enters service
  - make *Faction* a base class: rename *faction_t* to *FactionId*, but leave it and constants in *NbTypes.h*; move *strFaction* into *Faction*
  - each *Faction* contains a scheduling condition variable
  - thread waits on its faction's conditional variable instead of acquiring the RTC lock
  - before waiting, mark the thread as ready (not blocked) and increment the faction's count of ready threads
  - after waiting, reset the flag associated the with condition variable, decrement the faction's count of ready threads, and mark the thread as running
  - implement scheduler in a new thread or by modifying *InitThread*
  - scheduler signals a faction's condition variable to let one of its threads run
  - scheduler uses timewheel to decide which faction will run next
  - when a thread is about to pause or perform a blocking operation, it interrupts the scheduler so that the next faction can be chosen
  - when a thread in payload faction or lower runs preemptably, reduce its priority to *LowPriority* so that it will not contend with unpreemptable threads
  - scheduler gives time to preemptable threads by not signalling the condition variable of any faction
  - track and display time spent in each faction during the last short interval (5 seconds)

✓ use atomic operations for *TraceBuffer* and *Debug::ft* locks

x support message priorities in *Thread.EnqMsg* [no: if a thread needs to prioritize work, it should define work queues, similar to *InvokerThread*]

x implement a lightweight lock based on Windows' CRITICAL_SECTION [no: it is Windows specific and, unlike a mutex, it is not recovered if a thread exits without releasing it]

x support thread-specific storage [no: *Thread* subclasses can define their own data members, using dynamic allocation if there is not enough space in the thread *ObjectPool* block]

✓ enhance *SysMutex* to track the thread that is holding the mutex

✓ implement leaky bucket counter for run-to-completion timeout

  - addresses the problem that an RTC timeout can occur if a *preemptable* thread runs during time allotted to an unpreemptable thread: without this fix, a preemptable thread must either run briefly or have a lower priority than all unpreemptable threads

  - when testing on Windows, other threads are clearly running on the same processor, particularly when the executable is started
- ✓ vary RTC timeout according to *WarpFactor*
- x reduce frequency at which *InitThread* interrupts *RootThread* [no: adds complexity, and the current overhead is very low]
- ✓ convert thread statistics to true statistics (want to override output to retain tabular format; support an *Accumulator* subclass of *Statistic*)
- ✓ create *Debug.Reset* from code at the top of *Exception.ctor* and also call it from signal handlers
- ✓ block trap request if currently handling a trap, which raises the prospect of having to queue signals (decided not to queue them, but a signal that forces thread to exit overrides one that doesn't)
- ✓ implement forced exit (raising a final signal)
- x add *Thread.HenqMsg* for a synchronous reply: suppress *interrupt* if queuing an asynchronous message when waiting for a synchronous reply; discard the message if thread is not waiting for a synchronous reply [no: do not want to support synchronous messaging]
- ✓ change thread functions to *static* if they can only be invoked on the running thread
- ✓ in lab, only have *RootThread* (*InitThread*) output a log unless asked to kill thread
- ✓ on ctrl-C or ctrl-Break (Ctrl-Fn-F12), kill running or locked thread (see *SetConsoleCtrlHandler*: registering this handler was unnecessary because signal handler receives *SIGINT* and *SIGBREAK*)
- ✓ support trace triggering when a thread receives a message (added to *Thread.DeqMsg*)
- x eliminate orphan handling by making *~Thread* private and relying on *Destroy* [no: this causes compile errors in subclasses, and it is also difficult to block the deletion of a singleton thread]
- x add a *GlobalAddress* equivalent for messaging to threads [no: input handler can simply deliver message to the thread that serves the IP port]
- ✓ messaging to threads: add *MsgBuffer* queue and use *Interrupt*
- ✓ make threads pooled objects (needed in order to use *MsgBuffer* for thread messaging)
- ✓ capture statistics for each thread and log context switches
- ✓ allow extra time running unpreemptable during recovery
- ✓ thread functions are still being invoked before thread is fully constructed, causing a trap: acquire RTC lock before invoking *Thread.Start*
- ✓ add maximum time running unpreemptable as high watermark for each thread
- x catch unsupported signals passed to *raise* [no: problem is that kernel has a per-thread table that maps a signal to a handler, so there isn't space to register a handler against an unsupported signal]
- ✓ investigate *terminate* and *set_terminate* [per-thread; default terminate calls *abort*]
- ✓ call *MakeUnpreemptable* at top of Thread.Start; free RTC lock when returning or exiting abnormally
- ✓ enhance exception handling
  - ✓ see "exception handling...during debugging" in VS Help (Debug>Exceptions... menu)
  - ✓ use *_set_se_translator* to map Windows Structured Exceptions to C++ exceptions
  - ✓ use compiler option /EHa to catch Structured Exceptions (see "/EH" in VS Help)
  - x use *_set_purecall_handler* to handle pure virtual function calls [no: provide a default implementation for each pure virtual function]
- x detect thread that blocks when unpreemptable [unnecessary, because RTC timeout will kill it]
- ✓ cause restart if critical thread (add a function to determine this) cannot be recreated
- ✓ implement test for RTC timeout: >recover loop
- ✓ implement watchdog for RTC timeout in *InitThread*
- ✓ use same priority in all but *Watchdog* and *System* factions
- ✓ support thread trace events
- ✓ if using thread-specific store, *Thread.Start*--not the constructor--must create and set it because these calls act on the *running* thread, whereas the constructor runs in the *parent* thread's context [thread subclasses can add their own data, so decided not to use thread-specific store]
- ✓ kill thread if per-thread trap threshold is exceeded (use *LeakyBucketCounter*)

OPERABILITY

- ○ convert software logs to the string version of *Debug::SwErr* when this would provide better information
- ○ periodically close report files and start new ones
- ❖ log framework enhancements
- ❖ alarms (use during overload)
- ❖ distribution framework (control node)
- ❖ distributed data framework
- ❖ checkpointing framework
- ❖ failover

- ✓ *Debug::SwErr* should omit the stack trace for *Info* logs
- ✓ enhance *CfgParm* and subclasses to support configuration parameters that require a restart to be changed
- ✓ add *StatisticsThread* for periodically generating statistics reports and rolling over registers
- ✓ output logs directly if *LogThread* has not been created
- ✓ add *LogThread* for outputting logs
- ❖ OM framework
  - ✓ support peg counts, high and low water marks, groups, and rollovers
  - ✓ declare registers and groups as members and bind separately
  - ✓ support listing groups, displaying all OMs, or displaying a group
  - ✓ merge object pool and POTS OMs
  - ✓ add OM for number of objects recovered by object pool audit
  - ✓ add OMs for thread events
  - ✓ add OMs for context scheduling events
  - ✓ add OMs for internal/external messages sent/received by each factory, with longest for each
  - ✓ add OM for contexts created by each factory
  - ✓ add OM for timeouts sent
- ✓ add function traceback to software warning logs

CLI

- ➢ *CliThread*, although not hung, did not output prompt in response to multiple CRLFs (might have been caused by scheduling timeout, which causes all threads but one to disappear)
- ➢ in debug build, ctrl-C sometimes causes an unknown thread to exit (see Visual Studio *Output* window)
- o if RTC timeout is disabled, increase the priority of *CliThread* to ensure that a runaway thread can be killed
  - – may be desirable even when the RTC timeout is enabled
  - – raise priority and *SetPreemptable* just before reading the console
  - – determine if the input is a priority command that will cause *CliThread* to remain in this mode instead of reverting to normal priority and acquiring the RTC lock
- ? provide option to not echo commands (and their responses) when reading from a file
- o enhance symbols (see notes in *Symbol.cpp*)
- o enhance >if with *else* clause
- o make *CliText.text_* optional in a mandatory parameter
  - – preceding its *parms_* with *text_* is unnecessary when they must be present (however, it might prevent an ambiguous parse)
  - – *text_* is only needed if the parameter is optional (e.g. POTS facility parameters)
  - – could an application kludge it by flipping the use of *GetTextParm* and *GetTextParmRc* to treat mandatory *text_* as optional and vice versa?
- o spawn a thread to run testcases (add >run command; spawn non-console version of *CliThread*, which would no longer be a singleton; update pass/fail counts on original *CliThread*)
- o rework commands to begin with nouns instead of verbs (e.g. >trace start and >trace save instead of >start and >save trace)
- o support ";" as a command separator
- o allow tagged parameters to appear in any order
- o adopt Unix command syntax ("-<tag> <value>")

- ✓ make "\" an escape character and support string literals (text with embedded blanks or special characters)
- ✓ implement *CliCommandSet* so that commands can appear below the top level of a parse
- x console input enhancements (*CinThread*) [no: these are lower level capabilities of the platform]
  - – retrieval of previous commands (up arrow and down arrow to scroll through list)
  - – suspending and resuming console output (ctrl-S and ctrl-Q without carriage return)
  - – insert, delete, backspace, left arrow, right arrow, return, and escape for line editing
- ✓ register CLI symbols during initialization
- ✓ implement >symbols assign <sym> <parm> to save the value of a configuration parameter so that it can be subsequently restored
- ✓ test ctrl-C to abort in-progress CLI work and reenter *CliThread*
- ✓ support application-specific data by adding an array of pointers to *CliThread* (see *CliAppData*)
- ✓ modify commands that display pooled objects so that they return the number of objects found
- ✓ implement separate threads for *cout* and *cin*
- ✓ add option to >help to get verbose display of all commands in an increment
- ✓ implement >symbols set to support mnemonic symbols in CLI commands
- ✓ support nested >read (change *in_* to a stack)
- ✓ support CLI command inheritance for layering
- ✓ support tagged parameters, for example ">activate nnnnn cfd **dn=**nnnnn **to=**n" for tagging optional DN and timeout values
- ✓ check for the addition of duplicate command or string, as this causes it to be masked
- ✓ allow the rest of an input line to be executed as a command
- ✓ make "/" a comment character
- ✓ echo commands to output stream during >read
- x enforce that optional parameters follow mandatory parameters [was subsequently removed for >verify]

- ✓ add >read command for executing commands in a file
- ✓ allow commands to check for non-empty buffer after parameter parse; option to output “extra ignored”
- ✓ make “~” a skip character that results in the default value for an optional parameter

I/O

- o enhance socket functions to handle specific errors
- o convert to/from big endian when sending/receiving over IP
- o if message loss is an issue, support using a TCP connection for server-to-server communication (one per protocol; messages that would otherwise use UDP would share this TCP connection)
- o add calls to *IoLock* and *IoUnlock* where required
- o IPv6
- ? support out-of-band TCP (may involve *SIGURG* signal)
- ? use *SO_REUSEADDR* during error recovery (see http://stackoverflow.com/questions/14388706/socket-options-so-reuseaddr-and-so-reuseport-how-do-they-differ-do-they-mean-t)

- x discard retransmitted buffer if original buffer is still queued for output [no: would need to add message sequence numbers, and would not occur if connection was already established, which is what the interface simulates]
- ✓ TCP on outgoing protocols (use of *connect*; use of *accept* is already supported)
  - – add a virtual function to determine if a TCP socket is needed
  - – register socket with *TcpIoThread*
  - – socket is non-blocking, so *connect* "fails"
  - – save outgoing message until it can be sent
  - – poll socket to see when it is writeable
  - – article on TCP mistakes: http://tangentsoft.net/wskfaq/articles/lame-list.html
  - – article on sending small data segments over TCP: http://support.microsoft.com/kb/214397/EN-US
- ✓ provision protocols (*IpService*) instead of creating fixed IP ports
- ✓ support IP lookup functions in *SysSocket*: *winsock2.h* provides
  - ✓ *getaddrinfo*
  - x *gethostbyaddr* (deprecated: use *getnameinfo*)
  - x *gethostbyname* (deprecated: use *getaddrinfo*)
  - ✓ *gethostname*
  - ✓ *getnameinfo*
  - ✓ *getpeername*
  - x *getprotobyname* (not added)
  - x *getprotobynumber* (not added)
  - x *getservbyname* (not added)
  - x *getservbyport* (not added)
  - ✓ *getsockname*
- ✓ add statistics for each *IpPort*: messages sent/received, bytes sent/received
- x modify I/O threads to time out immediately on read when socket claims to have a packet waiting, since thread will otherwise hold RTC lock until a message arrives [no: this is *too* defensive]
- ✓ I/O thread should *Pause(0)* before *EnterBlockingOperation* to enable round-robin scheduling: if it doesn't do this, it can reacquire the lock almost immediately if its socket has a pending message, in which case it runs for a *long* time under heavy messaging load (use select to check if a message is pending: one issue is that you want to *Pause(0)* before *EnterBlockingOperation*, but only if you *don't* block, otherwise you get scheduled out *twice*)
- ✓ use *setsockopt SO_SNDBUF* and *SO_RCVBUF* to define size of send and receive buffers
- ✓ update *SysUdpSocket::MaxPacketSize* by calling *getsockopt* to find *SO_MAX_MSG_SIZE*
- ✓ use *ioctlsocket* to make receives blocking and sends non-blocking
- ✓ generate log if *IpBuffer.Send* times out: indicates that output threads are required

SESSIONS

- o implement a simple protocol stack (probably CIP over a quasi-SIP) for testing purposes
- o delay cleanup of context after a trap to focus on new work and avoid risk of retrapping: need a new context state (recovering) to block incoming messages and the recovery of subtended objects
- o track how much time each context is using to detect CPU hogs (e.g. a messaging loop between two contexts)
- o if peer PSM has a message queued on its context, *JoinPeer* should either fail or move the message
- o during *JoinPeer*, look at moving the entire protocol stack and then cloning it at its original location (it must be cloned anyway—the *Synch** functions—but moving the existing stack would eliminate the "overwrite if root SSM is a multiplexer" check in *PotsProfile.SetObjAddr*; furthermore, the stack would move to the edge, and the clone might only retain the upper layer)
- o support moving a PSM into its own context so that an SSM can idle itself even when a protocol requires an ack for a release message [can discard the ack as stale unless it releases a resource, such as a call reference]
- o implement a service for testing *ServiceSM.ProcessEvent*
- o implement commands for enabling and disabling a modifier
- o implement functions for replacing event handlers and triggers, which can be bound against several services (also move trigger's initiator queue); these objects are not yet unbound anywhere
- ? use member function pointers (fast delegates) to implement event handlers as SSM functions (see http://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible)
- ? support formal protocol state machines (PSMs)
  - subclass that implements *ProcessIcMsg* and *ProcessOgMsg* by delegating to event handlers
  - similar to the service model for states, events, event handlers
  - refactor *Service* into *Service* and *StateMachine*, with latter reusable for PSMs
- ? support message generation elements (MGEs) or some way of adding parameters at end of transaction
- ? support message deferral
  - specify the only PSM (PSM chain) from which a message is desired
  - must have a timer running on this PSM to provide a timeout
  - messages arriving on other PSMs are queued on context
  - when message arrives on target PSM, it is processed; deferred messages are handled at the end of the transaction unless deferral is restarted
  - add a virtual function to *ServiceSM* for screening messages before deferral (e.g. to allow a release message to be processed immediately)

- ✓ support protocol stacks (protocol 1 over protocol 2, as in ISUP over SIP)
- ✓ statistics for invoker pools
- x remove *PsmFactory* and *PsmContext* if a way to send a message (other than in response to one) cannot be found [use *SendToSelf* to get scheduled on the rare occasions when a message needs to be sent]
- ✓ at end of transaction, root SSM should invoke media gateway PSM last [would be done using PSM priority]
- ✓ define a "kill" message that can be sent from the CLI and that is sent by the default *InjectFinalMsg*
- x don't destroy an SSM in the Null state: MS call queried VLR in that state [it could query VLR in AO or AT state and trigger SAP *after* VLR response arrives]
- ✓ support multiplexer insertion/deletion
- ✓ support asynchronous modifier requests (suspending/resuming execution while on the initiator or modifier SSM queue)
- ✓ context trap leaves event orphaned for audit: queue events on SSM
- ✓ immediately delete a modifier that remains in the Null state after being invoked
- ✓ do not pass SAPs, SNPs, and SIPs to a modifier in the Null state [superseded by *idled_*]
- ✓ timer registry and PSMs both claim timers: change timer registry to simply audit timer queues
- ✓ enforce no state change at end *of AnalyzeMsgEventHandler*

MESSAGING

- define object pools for *IpBuffer* bytestreams to replace use of *Memory.Alloc*
- re-inject messages that are still queued on a context when it idles (a factory or protocol must be consulted before doing this)
- speed up *MsgPort.FindPeer* (currently searches through all ports)
- after disabling WiFi to mess up socket layer, IP address is lost, so *CanBypassStack* can return false, and then
  - timeout message gets sent over IP stack, so *PotsCallHandler* generates a lost message log because POTS header is absent [timeout message uses *SendToSelf*, which needs a way to force stack bypassing]
  - CWT has multiplexer with loopback address on NPSM; POTS call sends Facility message, bypassing stack because *rxaddr* is loopback address; reply from NPSM does not bypass stack because POTS UPSM still has the (now invalid) host address [perhaps *txaddr* should change to loopback address in this case]
- provide message inspection in debug mode: scan for messages that have a nil protocol or signal, that don't match the PSM's protocol, that contain illegal parameters, that are missing mandatory parameters...
- overload control enhancements
- babbling idiot controls

- x *SendFinalMsg* should use a *noexcept* function to allocate a message [no: throwing an exception during exception handling need not be fatal; the problem is throwing an exception while in a destructor]
- x limit the number of messages queued on a context; discard subsequent messages and notify source [no: overload controls now support discarding request-cancellation pairs and retransmissions]
- x have *Message.Relay* verify that the outgoing PSM supports the relayed message's protocol
- x support sub-parameters [no, this is simply a way to reduce usage of parameter identifiers]
- x initialize parameters with templates [no: can be done by message subclasses]
- ✓ manual inheritance for protocols (e.g. *GetSignal*, *GetParameter*, and iterators)
- x enforce read-only incoming messages [no: for an incoming message, *FindParm* would have to return a *const* pointer, this would force incoming and outgoing messages to be separate subclasses, which would make morphing mandatory for *Message.Relay*]
- x *Message.Relay* should return a message pointer in case the message has to be cloned [no: would only be cloned if it was saved, but it will remain saved anyway, and which PSM holds it should not matter]
- ✓ remove *initial_* and *final_* flags in *Signal*: they aren't used and don't apply to all protocols
- x add *Factory* method to *Message* for initializing header [no: different factories cannot share the same *Message* subclass]
- x if host address is lost, intra-processor messaging could survive by using loopback address [no: when IP stack is used deliberately, it is *intended* to simulate inter-processor messaging, which *should* fail in this case]
- ✓ enhance *Message.Send* so that intra-processor messages can be forced to use the IP stack
- ✓ display messages symbolically

TOOLS

- ○ trace tool enhancements
  - adjust constructor start times based on original start times of root and leaf constructors
  - add fence pattern to *TraceBuffer* to detect trampling by previous record's constructor
  - *startat* option: start tracing when a specific function is invoked, perhaps by a specific thread or by another specific function
  - *stopat* option: stop tracing when a specific function is invoked or when the "start at" function returns (as determined by its original stack depth)
  - refactor *BuffTrace* so that it can capture messages received by threads
- ○ testcase database: identifier, file, time of last execution, pass/fail status, functions invoked
- ○ session test environment: injecting a timeout message, discarding a message, synchronizing two messages
- ○ trace tool propagation (if a context is traced, then associated contexts get traced)
- ❖ trace tool networking (start trace tool on multiple nodes and send results to control node)
- ○ have MSC reuse context columns: put SSMs that ran sequentially on behalf of the same user in the same column, which might be possible if factories added a userid to the appropriate trace records
- ○ option to output trace as HTML, with hyperlinks between MSC messages and trace events
- ❖ TIMECALL equivalent (*TransTracer* sorting transactions by "call type"—a particular message sequence)
- ❖ SEGCT equivalent (*FunctionTracer* for code coverage: record functions invoked during a testcase; aggregate this data over testcases; would need CPU specific flow-of-control exception to capture jumps and returns)
- ❖ include stack variables in stack traces
- ❖ HTML-based code generator for
  - services: *Service*, *States*, *Events*, and *Triggers*; *EventHandler*, *Initiator*, and *ServiceSM* shells
  - protocols: *Protocol*, *Signals*, and *Parameters*; *InputHandler*, *Message* and *ProtocolSM* shells
  - *Factory* shell
- ○ Visual Studio plug-ins: see visualstudiogallery.msdn.microsoft.com
- ○ CMAKE: cross-platform builds
- ○ PurifyPlus (IBM) or Insure++ (Parasoft): memory leaks
- ○ Klocwork (Rogue Wave): static analysis
- ○ PC-Lint (Gimpel Software): static analysis

- ✓ implement trace variant that only records the number of times each function was invoked
- ✓ allow tracing to be stopped and restarted: add a "line break" record when this occurs
- ✓ add >testcase set recover to specify a file to execute after a testcase fails
- ✓ implement tool to
  - find deviations from coding guidelines
  - find read-only/write-only data
  - find unused functions
  - find missing/unnecessary *#includes*
  - analyze *#include* relationships
- x create a separate *SbIpBuffer* pool for *BuffTracer* so that it does not contend with payload applications
- x restrict tool usage to one user [of marginal value: only needed with multi-user *CliThread*; tool usage in field should be rare, and developers should not have to share platforms in the lab]
- x trace (creation/deletion) of IP buffers and contexts [of marginal value]
- x occasional MSC has only one context (*lines_*), but there should be at least two [can occur after a trap]
- x option to use different file suffixes when outputting subset of trace records [of marginal value]
- x option to output trace, function profile, and MSC in one file [of marginal value, and may want to see them side by side]
- x add per-thread trace buffer for preemptable threads [of marginal value because there isn't much to trace]
- ✓ implement a per-context trace buffer for message history that is included in dumps and logs
- ✓ DISPCALL equivalent (*Context.Dump* function for application use): applications can use *Logs.Spool*

- ✓ option to include/exclude factions in a trace
- ✓ option to define files that are read before and after >testcase
- ✓ initialize debug flags based on *InitFlags::TraceInit*
- ✓ constructors display in “reverse order” because of calls up the class hierarchy: resort them
- ✓ compiler-generated function is missing from trace just before a delete when the destructor chain is followed by an invocation of *operator delete*
- ✓ include only selected threads in function trace output (use >include and >exclude)
- ✓ add profiler for *FunctionTracer* (functions sorted by number of invocations, time spent in them, or name)
- ✓ have >verify track each factory’s messages separately to eliminate irrelevant ordering dependencies
- ✓ when running a testcase, overwrite files generated during its previous run
- ✓ implement a command to disable >verify (allows script to be used purely to set up a testcase)
- ✓ MSC shows a POTS shelf originator as external because *CliThread* sends messages on behalf of the shelf [partially fixed by adding a factory to the MSC when a message is not sent from a context, but shelf is a recipient during terminations, and the sessions that it receives and initiates can’t be distinguished]
- ✓ MSC assumes that each address had one peer, but this is false after inserting a multiplexer
- ✓ lock trace buffer early during *Debug.ft*, else stack overflow occurs when retracing a function that is reinvoked before constructing the original function’s record
- ✓ have message tracer capture messages sent by POTS shelf when there is no running context
- ✓ exclude *RootThread* and *InitThread* from traces, because they can block the trace buffer
- ✓ support tracing of initialization (*InitFlags::TraceInit*)

MEDIA, BASIC CALL, AND POTS

- ○ create CIP testcases that correspond to existing testcases
- ○ finish CFX testcases and add CFU to invalid, CFU loop, and A-B(CFU)-A(CFB)-C
- ○ improve testcase comments for CWT
- ? nullify incoming and outgoing media info when PSM enters Idle state, and block subsequent changes to these fields: problem is that this can be too late on the outgoing side; in any case, the PSM will be deleted at the end of the transaction, and any listener would then connect silence, *assuming that its outgoing message has not yet been sent*
- ○ support timeswitch ports and DNs in >include, >exclude, and >clear trace commands
- ○ support POTS data in >status command
- ○ create virtual *DnProfile* as base class for *PotsProfile* and a new *CipProfile* (for testing)
  - – profile will contain the factory identifier and DN that are placed in the route selector
  - – need to evolve >register and >subscribe commands to handle DN subclasses
- ○ check that basic call functions also apply to proxy and trunk calls
- ○ check that identifiers (events, event handlers, and triggers) defined in *BcSessions.h* are used
- ? remove *Event* from event names: for example, replace {<:a*Bc:a+}Event> with \1
  - – rename abstract classes first, so that they retain the *Event* suffix (e.g. *PotsCwtEventX*)
  - – change *EventStr* suffixes separately: not searching on end-of-word changes *EventHandler*
- ○ implement TWC, CXF (flash#1 = dial tone; flash#2 =conference or flipflop; flash#3 = release add-on or flipflop; onhook = transfer, release, or recall): includes conferencing
- ○ implement EXT (Extension Service): TBC with SPSM and multiple PPSMs
- ○ implement PSC (Preset Conference): TBC with non-POTS DN, multiple PPSMs, and conferencing
- ○ implement MMC (Meet-Me Conference): TBC with non-POTS DN, multiple PPSMs, and multiplexer
- ○ implement an offboard service (DN routing server for inter-processor CIP with multiple service nodes)
- ○ support pass/fail checks based on rxport, whether on-hook and idle, OM counts...
- ○ run POTS traffic with release build
- ❖ generic device pool (allocation with queueing, preemption, audit...)
  - – provide applications with a facade that communicates with pool and manages device PSM(s) and multiple devices (e.g. *n* conference circuits)

- ✓ CIP support for >inject and >verify
- ✓ modify testcases to read prolog and epilog files
- ✓ option to force CIP messages over IP stack while retaining *MsgHeader*
- x change POTS protocol to use *MsgHeader*, whether intra- or inter-processor: *PotsCircuit* has to remember PSM's address [no: should resemble a non-proprietary protocol]
- x provide control over whether messages to/from POTS shelf are external, interprocess, or intraprocess [no: should resemble an non-proprietary protocol]
- ✓ add reset capability to return circuit to initial state (and generate log if not in that state)
- ✓ change POTS shelf to retransmit offhook when receiving Release signal
- ✓ automated testing
  - – at testcase end: save CLI transcript, trace, and MSC in files that contain testcase name; include OMs
  - – at testcase start: clear buffer; set tools ftmc on; include all on; start
  - – implement testcase command that takes testcase name
- ✓ implement >verify command (purge outgoing message after checking it)