Robust Services Core

Software Overview

Release v0.28.0

June 6, 2022

TABLE OF CONTENTS

INTRODUCTION

The **Robust Services Core** (**RSC**) provides an infrastructure for developing robust C++ applications. It also provides tools for debugging and for the static analysis of C++ software.

RSC is designed for embedded, CLI-based applications. It allows software components to run in a distributed network, communicating with asynchronous protocols. A framework designed for these types of applications increases productivity and helps to assure the production of high-quality software with scalable capacity.

Although RSC was influenced by the needs of telecom servers, it could also be used

- in centralized servers, such as web servers
- in clients that communicate with such servers
- in distributed, embedded systems

RSC defines

- base classes that are subclassed to build applications,
- an object model for constructing applications from these classes, and
- key collaborations between the objects in the model.

RSC is implemented in three C++ namespaces that also correspond to static libraries:

- **NodeBase** contains classes that apply to all standalone nodes.
- **NetworkBase** contains classes for interprocessor messaging.
- **SessionBase** contains classes for session-oriented applications. Such applications consist of state machines driven by protocols.

There are also two libraries that contain test tools, namely **NodeTools** and **SessionTools**. These tools are packaged separately because they would not be deployed in an external release. This is also the case with **CodeTools**, which contains the C++ static analysis software.

A number of additional namespaces, which again correspond to static libraries, implement a POTS (Plain Ordinary Telephone Service) call server:

- **MediaBase** contains classes for SessionBase applications that also control media streams.
- **CallBase** contains classes for call processing applications.
- **PotsBase** contains software that is common to nodes that implement part of the POTS application.
- **AccessNode** contains software that interfaces to individual POTS clients.
- **ServiceNode** contains software for setting up calls between POTS clients.
- **OperationsNode, ControlNode,** and **RoutingNode** are unused but would contain software if the POTS application was divided as described in the "Distribution" section of the document *RSC Software Design*.

POTS was chosen because people are reasonably familiar with its specification. It therefore provides an accessible example of how to implement an application using RSC. Although its implementation omits details that would be found in a true call server, it is far from trivial. As such, it also serves as a vehicle for exercising RSC base classes.

BACKGROUND

After twenty years as a software architect at Nortel Networks, Greg Utas wrote *Robust Communications Software* and *A Pattern Language of Call Processing* to describe design patterns used in telecom servers. RSC implements these patterns, which are used in Nortel's GSM Mobile Switching Center (MSC), a call server for mobile networks. A brief history of this product should demonstrate the efficacy of these patterns.

In 1996, the MSC faced capacity and productivity challenges. It had been developed using Nortel's DMS switching platform, which was highly carrier grade. However, the platform's call handling software was not distributable, so capacity was limited to the throughput attainable by a single processor. Because the MSC supported heavyweight protocols, its throughput was 125K calls per hour, less than what customers wanted. At the same time, even more complex requirements were coming out of standards groups. These would further reduce capacity—assuming that they could be implemented at all. Nortel had entered the GSM market late and had therefore scrambled to get a product to market. The MSC's software had therefore been developed quickly, and it now had become difficult to evolve it to add new capabilities.

Addressing the MSC's challenges with incremental solutions was likely to fail, so the decision was to go all in. The MSC would keep the carrier-grade DMS platform but rearchitect its call handling (session processing) software. The new design would be implemented in an object-oriented language and would be distributable. Capacity would probably suffer because the platform was still uniprocessor. However, the redesign would improve productivity and, consequently, time to market. Nortel's platform group was also developing a shelf with multiple processor cards that could host distributed applications, which would eventually allow the capacity problem to be addressed.

Over a period of 18 months, the MSC's application software was rewritten while adding new features. Capacity declined by 8%, but productivity increased significantly. One developer remarked that development had become boring because it was obvious how to add new applications. In the late 1990's, one competitor had about four times as many MSC software developers on staff as did Nortel, yet both products had similar capabilities.

Once the redesigned MSC was deployed, a white paper was written to outline the steps needed to deliver a highly scalable, distributed product. This paper proved to be a roadmap for the next ten years. The platform group delivered the long-awaited processor shelf, allowing the product to eventually handle well over 2M calls per hour. The exact number was irrelevant because it far exceeded customer requirements. During this time, the product also added significant new capabilities, including VoIP, SIP, and H.248.

In 2009, Nortel declared bankruptcy. A competitor eventually bought the GSM MSC business unit. The goal of such a purchase is usually to acquire the customers, discontinue the acquired product, and sell the customers new equipment. The MSC's primary customer was AT&T. They were so pleased with the MSC that they could not be persuaded to replace it. Instead, they decided to fund its development group directly.

NODEBASE

This section provides an overview of NodeBase and NodeTools capabilities. In almost all cases, each header file defines a single class, which is implemented in a .cpp file having the same name. Consequently, names in italics are usually both class names and file names (.h and .cpp).

OPERATING SYSTEM ABSTRACTION LAYER

- *SysDecls:* low-level definitions (such as handles for native OS objects)
- *SysTypes:* low-level definitions that are platform independent
- *SysMemory:* low-level memory allocation, deallocation, and protection
- *SysHeap:* wrapper for default heap (objects derived from *Permanent* or a class not for a specific memory type)
- *SysThread:* wrapper for native threads and scheduler functions (such as the running thread)
- *SysSignals:* the POSIX signals supported on the platform
- *SysStackTrace:* stack disassembly (to obtain the chain of function calls on a thread's stack)
- *SysConsole:* console input, output, title, and minimization
- *RscLauncher:* launches RSC when support for automatic rebooting is required

The following have been converted to C++ STL classes that were unavailable when they were originally developed, so are now platform independent:

- *SysMutex:* locks for critical sections
- *SysLock:* low-overhead version of *SysMutex*
- *SysFile:* reading, writing, and finding files

RSC currently runs on both Windows and Linux. It should also be fairly easy to port it to other platforms, such as VxWorks. When porting to a new platform, the *Sys** header files should not require modification. The exception is *SysDecls.h* (see above). With regard to *.cpp* files, those having the same name as a *.h* file are platform independent and should not require modification. What need to be replaced are the platform-specific files, which in the case of Windows have names that end in *.win.cpp*.

Thought was given to defining the O/S abstraction layer in its own namespace, below NodeBase. However, the abstraction layer remained part of NodeBase so that it could use low level capabilities that are not platform specific, such as the *Object* class and function tracing, which allow it to better integrate with RSC applications and debugging tools.

TEMPLATES

- *Allocators*: for use with an STL container that is a member of an object that does not use the default heap
- *Array*: similar to *std::vector*, but with efficient erasure when the order of elements is unimportant
- *Singleton:* for singleton objects
- *Q1Way* and *Q1Link:* for one-way queues
- *Q2Way* and *Q2Link:* for two-way queues

- *Registry* and *RegCell:* for registering and accessing objects using numeric identifiers

TIME

- *Duration*: time intervals
- *SteadyTime*: monotonic time (timestamps)
- *SystemTime:* calendar and time-of-day time

OBJECTS

- *Memory:* provides memory types that have different persistence and protection characteristics
- *Base:* base class for simple objects that are typically members of other objects
- *Object:* base class for non-trivial objects
- *Class:* base class for non-trivial classes (to eventually support serialization/deserialization)
- *ClassRegistry:* registry for concrete subclasses of *Class*
- *Temporary:* base class for objects whose heap is freed during all restarts
- *Dynamic:* base class for objects whose heap is freed on cold and reload restarts
- *Persistent:* base class for objects whose heap is only freed during reload restarts
- *Protected:* base class for objects that are write-protected and whose heap is only freed during reload restarts
- *Permanent:* base class for objects that survive all restarts (uses default heap)
- *Immutable:* base class for objects that survive all restarts and that are write-protected
- *Singletons:* registry of singletons
- *ObjectPool:* base class for pools from which objects are allocated
- *ObjectPoolRegistry:* registry for concrete subclasses of *ObjectPool*
- *ObjectPoolAudit:* thread for recovering an orphaned *Pooled*
- *Pooled:* base class for objects allocated from an *ObjectPool*
- *PooledClass:* base class for classes whose objects are derived from *Pooled*
- *Heap:* base class for *SysHeap* and *BuddyHeap*
- *BuddyHeap:* heap for objects derived from *Temporary*, *Dynamic*, *Persistent*, *Protected,* and *Immutable*
- *SlabHeap:* expandable heap for objects derived from *Pooled* and whose *ObjectPool* uses *MemSlab*
- *HeapCfg:* for configuring the size of heaps other than the default heap
- *CallbackRequest:* base class for supporting callbacks

CONFIGURATION PARAMETERS

- *CfgTuple:* holds the key-value pair associated with a configuration parameter
- *CfgParm:* base class for configuration parameters
- *CfgIntParm:* base class for integer configuration parameters
- *CfgBitParm:* base class for *CfgBoolParm* and *CfgFlagParm*
- *CfgBoolParm:* base class for boolean configuration parameters
- *CfgFlagParm:* base class for flag configuration parameters (a bit in a word)
- *CfgStrParm:* base class for string configuration parameters

- *CfgParmRegistry:* registry for concrete subclasses of *CfgParm*

Configuration parameters define the node's environment and are read from the file *element.config* during initialization.

THREADS

- *Thread:* base class for threads
- *ThisThread:* functions that apply to the running thread
- *Gate:* interuptable condition variable
- *Daemon:* base class for recreating threads that exited because of errors
- *DaemonRegistry:* registry for concrete subclasses of *Daemon*
- *MsgBuffer:* base class for intraprocessor messages between threads
- *Deferred:* base class for work to be executed when a timeout or other event occurs
- *DeferredRegistry:* registry for concrete subclasses of *Deferred*
- *DeferredThread:* notifies work items in *DeferredRegistry* when their timeout occurs
- *FunctionGuard:* stack object for automatically invoking a function's conjugate when exiting from a scope
- *MutexRegistry:* registry for instances of *SysMutex*
- *ThreadRegistry:* registry for concrete subclasses of *Thread*
- *ThreadAdmin:* thread configuration parameters and statistics
- *PosixSignal:* base class for POSIX signals
- *PosixSignalRegistry:* registry for concrete subclasses of *PosixSignal*
- *Exception:* base class for exceptions
- *ElementException:* exception thrown to cause a restart
- *AllocationException:* exception thrown when allocation fails
- *AssertionException:* for an assert capability
- *SignalException:* exception thrown to handle a *PosixSignal*
- *SoftwareException:* for application exceptions

INITIALIZATION

- *InitFlags:* flags that enable debugging during initialization
- *Restart:* tracks system status and defines types for restarts
- *Module:* base class for initializing a unit of software (one-to-one with a static library and in most cases a C++ namespace, although nothing prevents their use at a more granular level)
- *ModuleRegistry:* registry for concrete subclasses of *Module*
- *RootThread:* created by *main*'s thread; creates *InitThread* and acts as an initialization and scheduling watchdog
- *InitThread:* initializes the system by invoking the modules in *ModuleRegistry*; controls thread context switching
- *main.cpp:* creates leaf *Module* subclasses and *RootThread*
- *MainArgs*: saves and provides access to *main*'s command line arguments

STATISTICS

- *Statistics:* base classes for peg counts and high and low watermarks
- *StatisticsGroup:* base class for grouping related statistics
- *StatisticsRegistry:* registry for concrete subclasses of *StatisticsGroup*
- *StatisticsThread:* thread to generate statistics reports and roll over statistics at fixed intervals

LOGS AND ALARMS

- *Alarm:* class for defining and raising alarms
- *AlarmRegistry:* registry for instances of *Alarm*
- *Log:* class for defining and generating logs
- *LogBuffer:* circular buffer for logs generated at run-time
- *LogBufferRegistry:* registry for instances of *LogBuffer*
- *LogGroup:* class for grouping related logs, with registry for instances of *Log*
- *LogGroupRegistry:* registry for instances of *LogGroup*
- *LogThread:* thread for sending logs to a file (and, in a debug load, the console)

NODES

- *Element:* information about this node

COMMAND LINE INTERFACE

The following support the command line interface:

- *CliParm:* base class for CLI parameters (arguments to a CLI command)
- *CliIntParm:* base class for integer parameters
- *CliBoolParm:* base class for boolean parameters
- *CliCharParm:* base class for single-character parameters
- *CliPtrParm:* base class for pointer parameters
- *CliTextParm:* base class for text parameters (an arbitrary string or one from a list)
- *CliText:* base class for strings that are followed by parameters
- *Symbol:* base class for symbols (strings that map to integer constants)
- *SymbolRegistry:* registry for concrete subclasses of *Symbol*
- *CliCommand:* base class for CLI commands
- *CliCommandSet:* base class for a group of related CLI commands
- *CliIncrement:* registry for a group of related CLI commands
- *CliRegistry:* registry for concrete subclasses of *CliIncrement*
- *CliThread:* parses input, invokes commands, and displays output
- *CliBuffer:* used by *CliThread* to parse a command line
- *CliCookie:* tracks the current location in the parameter tree while parsing a command line

- *CliStack:* used by *CliThread* to track active increments
- *CliAppData:* provides application-specific storage for *CliThread*

These classes improve productivity by providing a high-level interface for command line parsing. At the same time, they promote a common look and feel for CLI commands, one which also ensures that each command and parameter is documented in the help command.

INPUT/OUTPUT

Input and output are offloaded to separate threads with a view to eventually supporting a remote console and remote files:

- *CoutThread:* interface for writing to the console
- *CinThread:* interface for reading from the console
- *FileThread:* interface for writing to a file
- *Formatters:* functions that format data being sent to the console or a file
- *StreamRequest:* subclass of *MsgBuffer* for sending an *ostringstream* to *CoutThread* or *FileThread*

DEBUGGING

- *ToolTypes:* definitions for debugging tools
- *Tool:* base class for trace tools
- *Debug:* interface for tracing functions and generating software logs
- *TraceRecord:* base class for recording events during debugging
- *TimedRecord:* base class that records the time and thread associated with an event
- *FunctionTrace:* records a function call
- *MemoryTrace:* records a memory allocation/deallocation
- *ObjectPoolTrace:* records the allocation/deallocation of a block in an *ObjectPool*
- *TraceBuffer:* circular buffer for *TraceRecords*
- *TraceDump:* displays the records in a *TraceBuffer*
- *FunctionStats:* statistics about a function’s invocations
- *FunctionProfiler:* sorts and displays all *FunctionStats*

INTERNAL

- *LeakyBucketCounter:* tracks how often an event occurred during an interval
- *NbTypes:* defines frequently used types to reduce compile-time dependency on the interfaces that would otherwise define them
- *NbModule:* initializes NodeBase
- *NbLogs:* logs for NodeBase
- *NbPools:* object pools for threads and message buffers
- *NbDaemons:* daemons for NodeBase

- *NbIncrement:* CLI increment for NodeBase
- *NbCliParms:* CLI parameters for NodeBase types
- *NbAppIds:* identifiers for application classes derived from NodeBase base classes
- *NbTracer:* traces specific threads or factions
- *NtModule:* initializes NodeTools
- *NtIncrement:* CLI increment for NodeTools
- *NtTestData:* subclass of *CliAppData* for NodeBase testing

NETWORKBASE

For interprocessor messaging, RSC provides classes that support IP applications. These classes reside in their own static library (the *nw* directory) and namespace (NetworkBase) that is built on top of the primary NodeBase library (the *nb* directory).

BASIC TYPES

- *NwTypes:* basic types for NetworkBase
- *SysIpL2Addr:* an IP layer 2 address
- *SysIpL3Addr:* an IP layer 3 address

SOCKETS

- *SysSocket:* base class for a socket
- *SysUdpSocket:* a UDP socket
- *SysTcpSocket:* a TCP socket

SERVICES

- *IpService:* base class for an application that uses an IP protocol
- *IpServiceCfg*: base class for enabling or disabling an *IpService*
- *IpServiceRegistry:* registry for concrete subclasses of *IpService*
- *TcpIpService:* base class for an *IpService* that uses TCP
- *UdpIpService:* base class for an *IpService* that uses UDP

PORTS

- *IpPort:* base class for an *IpService* running a specific IP port
- *IpPortRegistry:* registry for concrete subclasses of *IpPort*
- *UdpIpPort:* base class for an *IpPort* that uses UDP
- *TcpIpPort:* base class for an *IpPort* that uses TCP

MESSAGES

- *InputHandler:* base class for passing messages from an *IoThread* to an application
- *IpBuffer:* base class for UDP and TCP messages
- *ByteBuffer:* buffers for *IpBuffer* message contents

THREADS

- *IoThread:* base class for I/O threads
- *UdpIoThread: IoThread* for services that use UDP
- *TcpIoThread: IoThread* for services that use TCP

INTERNAL

- *NwModule:* initializes NetworkBase
- *NwLogs:* logs and alarms for NetworkBase
- *NwPools:* object pools for IP messages
- *NwDaemons:* daemons for NetworkBase
- *LocalAddrTest:* tests that the local host address can send and receive messages
- *NwIncrement:* CLI increment for NetworkBase
- *NwCliParms:* CLI parameters for NetworkBase types
- *NwTracer:* traces specific IP addresses or ports
- *NwTrace:* trace records for socket events

SESSIONBASE

This section provides an overview of SessionBase and SessionTools capabilities. Again, names in italics are usually both class names and file names (.h and .cpp).

PROTOCOLS

- *Protocol:* base class for defining protocols (a set of signals and parameters)
- *ProtocolRegistry:* registry for concrete subclasses of *Protocol*
- *Signal:* base class for defining signals (a message type) within a *Protocol*
- *Parameter:* base class for defining parameters within a *Protocol*
- *Message:* base class for parsing/building the contents of an *SbIpBuffer*
- *MsgPort:* bottom layer in a protocol stack; manages addresses for sending and receiving messages
- *ProtocolLayer:* base class for a layer in a protocol stack
- *ProtocolSM:* base class for an individual protocol state machine in a protocol stack
- *SbIpBuffer:* base class for message payloads
- *GlobalAddress:* an IP layer 3 address plus a *LocalAddress*
- *LocalAddress:* identifies a *Factory* and, if one exists, a *ProtocolSM*
- *MsgHeader:* header for SessionBase messages

SERVICES

- *Service:* base class for defining services (a set of states and event handlers)
- *ServiceRegistry:* registry for concrete subclasses of *Service*
- *State:* base class for defining a service's states (event to event handler mappings)
- *Event:* base class for defining a service's events
- *EventHandler:* base class for defining a service's event handlers
- *RootServiceSM:* base class for service state machines (run-time instances)
- *AnalyzeMsgEvent:* event raised by a *ProtocolSM* for a *RootServiceSM*
- *ServicePort:* base class for a service's identifiers for its *ProtocolSMs*

SUPPLEMENTARY SERVICES

- *Trigger:* base class that allows *Initiators* to create modifiers (supplementary services)
- *Initiator:* base class for creating a modifier when a trigger occurs in its parent service
- *ServiceSM:* base class for modifier state machines
- *InitiationReqEvent:* event raised to start a modifier
- *AnalyzeSapEvent:* event giving a modifier the chance to alter its parent's behavior
- *AnalyzeSnpEvent:* event giving a modifier the chance to extend its parent's behavior
- *ForceTransitionEvent:* event allowing a modifier to execute an event handler in its parent's context

TIMERS

- *Timer:* instance of a timer running on a *ProtocolSM*
- *TimerRegistry:* registry containing all timers
- *TimerProtocol:* defines *TimeoutSignal* and a parameter for distinguishing timers
- *TimerThread:* services the *TimerRegistry*

RUN-TIME

- *Context:* base class for aggregating the objects used by an application's run-time instance
- *MsgContext:* a context for a stateless application (query-response protocol)
- *PsmContext:* a context for a single *ProtocolSM*
- *SsmContext:* a context for a *RootServiceSM* and one or more *ProtocolSMs*
- *Factory:* base class for creating the objects that run in a context
- *MsgFactory:* base class for creating objects in a *MsgContext*
- *PsmFactory:* base class for creating objects in an *PsmContext*
- *SsmFactory:* base class for creating objects in an *SsmContext*
- *FactoryRegistry:* registry for concrete subclasses of *Factory*
- *InvokerThread:* invokes application logic after a context receives a message
- *InvokerPool:* base class for a pool of invoker threads
- *InvokerPoolRegistry:* registry for concrete subclasses of *InvokerPool*

TLV PROTOCOLS

- *TlvProtocol:* base class for protocols whose messages use a type-length-value encoding
- *TlvParameter:* base class for a parameter in a *TlvProtocol*
- *TlvIntParameter:* base class for an integer *TlvParameter*
- *TlvMessage:* base class for building and parsing a message in a *TlvProtocol*
- *TextTlvMessage:* base class for a *TlvMessage* that maps to a text-based protocol

MESSAGE SEQUENCE CHARTS

- *MscBuilder*: builds a message sequence chart (MSC) from messages captured by a trace
- *MscAddress:* an address (a *RootServiceSM*, *ProtocolSM*, or *MsgFactory*) in an MSC
- *MscContext:* a context (a vertical line) in an MSC
- *MscContextPair:* a pair of communicating contexts (a horizontal line) in an MSC

INTERNAL

- *SbTypes:* defines frequently used types to reduce compile-time dependency on the interfaces that would otherwise define them

- *SbModule:* initializes SessionBase
- *StModule:* initializes SessionTools
- *SbLogs:* logs for SessionBase
- *SbPools:* object pools for *SbIpBuffers*, *ServiceSMs*, *ProtocolSMs*, *Messages*, *Timers*, and *Events*
- *SbDaemons*: daemons for SessionBase
- *SbIncrement*: CLI increment for SessionBase
- *StIncrement*: CLI increment for SessionTools
- *SbCliParms:* CLI parameters for SessionBase types
- *SbInputHandler:* input handler for SessionBase protocols
- *SbExtInputHandler:* input handler for external protocols received by SessionBase applications
- *SbInvokerPools:* concrete classes for SessionBase invoker pools
- *SbEvents:* concrete events defined by SessionBase
- *SbHandlers:* concrete event handlers defined by SessionBase
- *SbAppIds:* identifiers for application classes derived from SessionBase base classes
- *SbTestData: CliAppData* for SessionBase testing
- *SbTracer:* traces specific factories, protocols, signals, or services
- *SbTrace:* trace records for SessionBase activity
- *TestSessions:* contexts that inject messages to test SessionBase applications

PATTERNS

RSC supports some patterns within base classes, whereas the use of other patterns must be left to applications.

ROBUST COMMUNICATIONS SOFTWARE

This section summarizes how RSC uses the patterns described in *Robust Communications Software*.

SUPPORTED BY BASE CLASSES

Table 1. Patterns supported by base classes.

Pattern	Remarks
Object Class	*Object*
Singleton	*Singleton*
Registry	*Registry*
Object Pool	*ObjectPool*
Object Nullification	*Pooled*
Thread Class	*Thread*
Cooperative Scheduling	*Thread.Pause*
Half-Sync/Half-Async	*IoThread* separated from *InvokerThread*
Run-to-Completion Timeout	*InitThread.HandleInterrupt*
Run-to-Completion Cluster	*MsgPriority = IMMEDIATE*
Leaky Bucket Counter	*LeakyBucketCounter*
Stack Overflow Protection	*Thread.StackCheck*
Safety Net	*Thread.HandleSignal, Thread.Start, Thread.TrapHandler*
Initialization Framework	*Module*
Write-Protected Memory	*Protected* and *Immutable*
Escalating Restarts	*ModuleRegistry*
TLV Message	*TlvMessage*
Parameter Typing	*TlvMessage.AddType*
Parameter Fence	*TlvMessage.ParmFencePattern*
In-Place Encapsulation	*IpBuffer*
Stack Short-Circuiting	*Message.Send*
Message Cascading	*Message.Retrieve*
Message Relaying	*Message.Relay*
Callback	*CallbackRequest* (should be used sparingly)
Finish What You Start	*MsgPriority*
Discard New Work	*InvokerPool.RejectIngressWork, Factory.ScreenIcMsgs*
Configuration Parameters	*CfgParm* and its subclasses
Logs	*Log, LogBuffer, LogGroup, LogThread*
Alarms	*Alarm*
Operational Measurements	*Statistics* and related classes
Software Targeting	*Sys** operating system abstraction layer
Run-Time Flags	*Debug.SwFlags_*
Software Error Log	*Debug.SwLog*
Software Warning Log	*Debug.SwLog*

Object Dump	*Object.LogSubtended*
Flight Recorder	*LogBuffer* and `logs*` files
Object Browser	*NbIncrement*, *SbIncrement*
Function Tracer	*Debug.ft, FunctionTrace*
Message Tracer	*BuffTrace*
Transaction Profiler	*TransTrace*
Thread Profiler	*SchedCommand*
Function Profiler	*FunctionProfiler*

Table 2. Patterns to be supported by base classes.

Pattern	**Remarks**
Object Template	by *Class*; rarely required
Quasi-Singleton	by *Class*; rarely required
Object Morphing	by *Class*; rarely required
Proportional Scheduling	by *Thread*
Reliable Delivery	by TCP-related classes; implementation is incomplete
Symmetric Multi-Processing	treat each CPU as an independent node
Shared Memory	treat each executable as a logical node
Binary Database	
Parameter Template	by *TlvMessage*
Load Sharing	support as a node maintenance strategy
Cold Standby	support as a node maintenance strategy
Warm Standby	support as a node maintenance strategy
Object Checkpointing	by *PooledClass* and *Pooled* (serialization)
Hitless Patching	
Rolling Upgrade	support as a node maintenance procedure
Maintenance	by *cn* directory and components in *an, on, rn,* and *sn* directories

USED BY IMPLEMENTATION

These patterns cannot be supported by base classes but are used in the implementation of RSC or the POTS call server.

Table 3. Patterns used within implementation.

Pattern	**Remarks**
Flyweight	ubiquitous
Polymorphic Factory	*Factory*
Embedded Object	*LeakyBucketCounter*
Thread Pool	*InvokerPool*
Half-Object Plus Protocol	*CipProtocol*, *BcSsm*, *BcPsm*
Defensive Coding	ubiquitous
Audit	*ObjectPoolAudit*
Watchdog	*RootThread*, *InitThread*
Heartbeating	*InitThread* to *RootThread, Daemon*
No Empty Acks	*CipProtocol*, *PotsProtocol*
Throttle New Work	application-specific

NOT IMPLEMENTED

Table 4. Unimplemented patterns.

Pattern	**Remarks**
Cached Result	application-specific
Low-Order Page Protection	platform-specific
User Spaces	application-specific; only recommended for logical nodes
Message Attenuation	application-specific
Eliminating I/O Stages	platform-specific
Prefer Push to Pull	application-specific
Polygon Protocol	application-specific
Ignore Babbling Idiots	to be implemented by *PotsCircuit*
Application Checkpointing	application-specific
Memory Checkpointing	platform-specific
Virtual Synchrony	not recommended
Hot Standby	platform-specific
Protocol Backward Compatibility	application-specific
Object Reformatting	application-specific
Hitless Upgrade	platform-specific
Heterogeneous Distribution	application-specific
Homogeneous Distribution	application-specific
Hierarchical Distribution	application-specific
Tracepoint Debugger	platform-specific
Set the Capacity Benchmark	not applicable (a planning technique)
Conditional Compilation	not recommended for software optionality

A PATTERN LANGUAGE OF CALL PROCESSING

This section summarizes how RSC uses the patterns described in *A Pattern Language of Call Processing*. The RSC terminology is sometimes different, so here is a mapping:

A Pattern Language of Call Processing	**RSC**
AFE	*ServiceSM*
Agent	*Factory*
feature	service
FSM	*Service*
PFE	*Initiator*
PFQ	*Trigger*
Transactor	*Context*

SUPPORTED BY BASE CLASSES

Table 5. Patterns supported by base classes.

Pattern	**Remarks**
State Machine	*Service*, *State*, *Event*, *EventHandler*, *ServiceSM*
Run to Completion	*InvokerThread*
Separation of Call Halves	*CipProtocol*, *CipPsm*
Agent Factory	*Factory* and its subclasses
Message Preservation	*Message.Save*
Separation of Basic Calls and Features	*RootServiceSM* and *ServiceSM*
FSM Observer	*AnalyzeSapEvent* and *AnalyzeSnpEvent*
FSM Model	classes defined in *BcSessions.h*
Hold Outgoing Messages	*ProtocolSM.EndOfTransaction*
Call Multiplexer	*ProtocolSM.JoinPeer* and *ProtocolSM.DropPeer*
Surrogate Call	classes defined in *ProxyBcSessions.h*
Call Distributor	classes defined in *ProxyBcSessions.h*
Run to Completion (extended)	*MsgPriority = IMMEDIATE*
Context Preservation	*Event.SaveContext*
Event Handler Visitor	*ForceTransitionEvent*
PFE Chain of Responsibility	*Initiator.GetPriority*
Initiation Observer	*InitiationReqEvent*

NOT IMPLEMENTED

These patterns could easily be supported if required.

Table 6. Unimplemented patterns.

Pattern	**Remarks**
Feature Networking	define a new *CipParameter* subclass
Connection Observer	enhance *MediaPsm*
Parameter Database	enhance *BcSsm*
Multiplexer Chain of Responsibility	define a priority for *RootServiceSMs*
Sibling Observer	define a new *Event* subclass and *ServiceSM* functions to pass it from one sibling to another