Robust Services Core

Session Processing

Version 2.0

November 27, 2017

CONTENTS

INTRODUCTION

This document describes SessionBase, the library for implementing services in RSC (Robust Services Core). As a framework specifically oriented towards building stateful services that communicate using protocols, SessionBase significantly increases designer productivity and helps to ensure the delivery of high quality software of acceptable capacity.

SessionBase defines

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

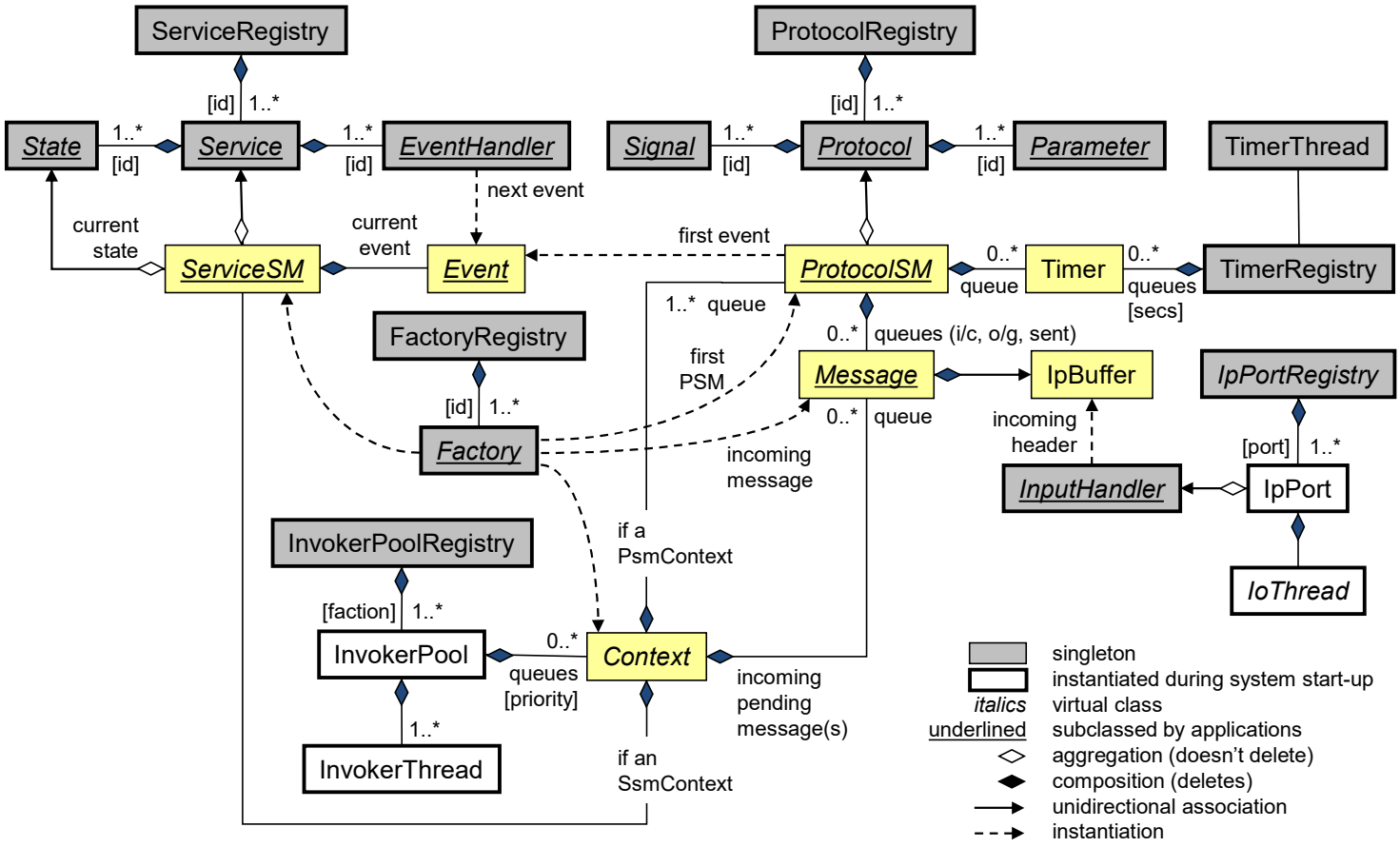
SessionBase is intended for control plane applications. Its object model is rather heavyweight for data plane applications, where it could cause an unwarranted performance overhead.

This document is intended to be a comprehensive *overview* rather than comprehensive. For further details, see the comments in .h files. The POTS application, implemented in the libraries MediaBase, CallBase, and PotsBase, provides an example of a moderately complex SessionBase application.

OBJECT MODEL

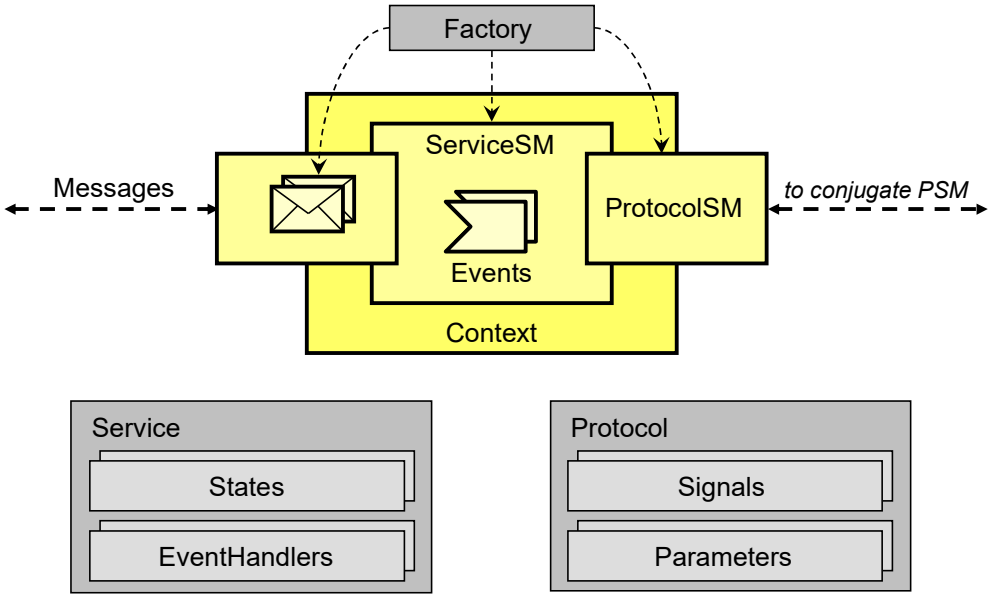
The SessionBase object model is shown in Figure 1. It defines orthogonal base classes that serve as building blocks for message-driven, stateful applications. The model is introduced here so that the reader can refer to it whenever one of its classes is mentioned.

Figure 1. SessionBase Object Model



An informal run-time view of the object model appears in Figure 2, which is from the perspective of an application because it only includes the primary classes that are subclassed to create services and protocols.

Figure 2. Application Run-Time View



Given that the goal is to implement message-driven, stateful applications, we need to look at state machines and messages at the next level of detail.

STATE MACHINES

The following components are required to define a state machine:

- A set of **States**. A state machine is always in some state that determines its behavior.
- A set of **Events**. Events are inputs to the state machine and arise in three ways:
  - When a message arrives.
  - When a timer expires.
  - When one event follows another to decompose work into a sequence of steps.
- A set of **Event Handlers**. When event arrives in a state, some event handler processes it.

Each run-time instance of a state machine requires its own member data so that, at a minimum, it can save its current state. The state machine instance reacts to events that are created at run time.

Other parts of a state machine, however, can be shared by all of its instances. States can be shared because whenever a state is presented with a given event, it always maps it to the same event handler. And if an event handler's arguments are the state machine's instance data and the event to be processed, it can also be reused, because it is simply an isolated function.

In the SessionBase object model, a **Service** consists of a set of states and a set of event handlers. These are the components that all of a state machine's run-time instances share. A **Service State Machine (SSM)** implements each run-time instance of a state machine. All SSMs of the same type share the same service.

Thus far, the object model consists of services, states, events, event handlers, and SSMs. These are the basic building blocks of state machines. But in complex systems, it is sometimes useful to create one state machine when specific conditions arise in another one. For example, a call waiting state machine is created when the busy condition arises in a basic call state machine. The new state machine augments or overrides the behavior of the first one, which avoids the pitfall of combining all applications into One Big State Machine.

Two new classes allow one state machine to observe and modify another's behavior. The first is a **Trigger**, which defines a condition that can arise in a state machine. Triggers are optional. But to make its behavior observable, a state machine must define at least one trigger. At run time, the state machine signals a trigger whenever the condition associated with the trigger arises. A state machine that wishes to observe another's behavior registers an **Initiator** with the appropriate trigger. When the trigger is signaled, it notifies the initiator, which may then create its SSM to augment or override the behavior of the SSM that it is observing.

MESSAGES

In most systems, a message is simply a buffer received by a thread. Within SessionBase, a **Message** is an object that wraps such a buffer. This allows applications to deal with buffers at a higher level of abstraction.

Each buffer invariably contains a message type, such as a specific type of request or reply. SessionBase uses **Signal** to refer to a message type. In many cases, the buffer also contains **Parameters** that are associated with the signal. Finally, each signal belongs to a **Protocol**. A protocol defines a set of signals, the order in which the signals may be sent and received, and the parameters that are mandatory or optional for each signal.

The protocol, signal, and parameters in a buffer are not actual objects. They are encoded in binary or text and may arrive from another processor, even a processor in another network. SessionBase assigns an identifier to each protocol and signal. If the buffer contains more than one parameter, as opposed to a single *struct*, then each parameter is also tagged with an identifier. These identifiers, for protocols, signals, and parameters, are used to access objects that are shared by many messages, in the same way that services, states, and event handlers are shared by many SSMs.

In the same way that an SSM is the run-time instance of a service, a **Protocol State Machine** (**PSM**) is the run-time instance of a protocol. Actually, two PSMs are needed to implement an instance of a protocol: one at each end of the logical **signaling channel** that connects the two state machines that are using the protocol to communicate. It is therefore more accurate to say that each PSM implements a protocol *role*, such as a client or server role.

Why are PSMs and SSMs separate objects? Couldn't they be combined? The answer is no, because many SSMs use more than one protocol at a time, and each of these protocols has its own state. A SIP SSM, for example, uses at least three PSMs: one to communicate with the subscriber, one to communicate with the far-end user, and one to communicate with a media gateway. Although a SIP SSM might define states that resemble those of a SIP PSM, this does not make them the same object. A PSM manages only its protocol. An SSM's role is different because it

coordinates the actions of multiple PSMs. It is therefore more oriented towards protocol interworking, and it exists to provide a service. In the case of a SIP SSM, the service is typically to coordinate the setup of a media stream with another user.

INSTANTIATION AND ASSOCIATION

Two more classes complete the object model. The first one, a **Factory**, instantiates objects at run time. The second one, a **Context**, associates them once they have been instantiated.

Before a SessionBase application can react to an input, a number of objects must exist. They include a message, a PSM, and an SSM. The message contents (the buffer) must somehow specify the subclasses of application objects that need to be created. The actual subclasses largely depend on the message's protocol, but basing the decision on protocol alone turns out to be inadequate. The message must therefore allow the subclasses to be specified in a more granular way. A *factory identifier* is used for this purpose, in the same way that identifiers are used for protocols, signals, and parameters. The factory identifier indexes a registry of factory objects, and the selected factory instantiates the necessary subclasses.

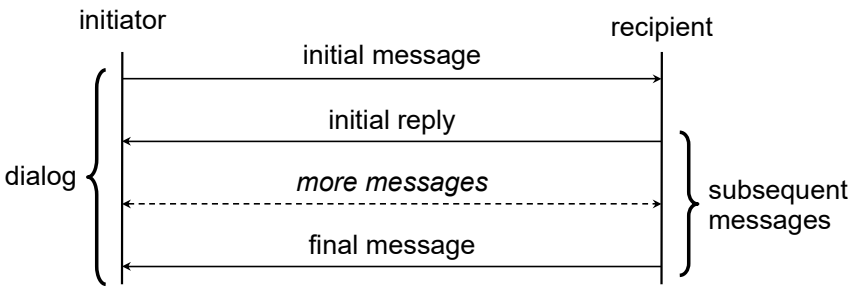
Once the appropriate objects have been created, they must remain associated until the state machine returns to the idle state. The second class, the context, provides this association by maintaining references to the SSM, the PSM(s), and the incoming message(s).

DIALOGS

The following concepts, illustrated in Figure 3, are central to SessionBase:

- A **dialog** is an exchange of messages within a specific protocol. A protocol may be based on an external standard, or it may be defined solely for internal use.
- An **initial message** is one that opens a dialog.
- A **final message** is one that closes a dialog.
- A **subsequent message** is any message other than an initial message.
- A dialog has two roles. The **initiator** sends the initial message and the **recipient** receives it. In a client-server model, the client is the initiator and the server is the recipient.
- An **initial reply** is the first message (if any) sent from the recipient to the initiator.

Figure 3. Terminology for Dialogs



A **connectionless protocol** consists of a request and a reply, which ends the dialog. It is therefore stateful only for the initiator, who awaits the reply. The recipient, however, need not save state information. The term **transaction processing** refers to the recipient's handling of this type of dialog.

A **connection-oriented protocol** manages a session, which is stateful for both participants. It has a setup phase (an initial message and initial reply), a data exchange phase ("more messages" in the figure), and a disconnect phase (a final message). The term **session processing** refers to implementing this type of dialog. All of the messages belong to the same dialog, and the session persists until the dialog ends.

Although SessionBase is designed for session processing, some stateful services also use connectionless protocols. SessionBase must therefore support transaction processing efficiently.

CONTEXTS

A context, which was introduced in "Instantiation and Association" on page 9, associates objects whose lifetime is determined by the dialog(s) that they support. SessionBase defines three types of contexts:

1. A **MsgContext** is used for transaction processing. The initial message is passed to a protocol-specific function that performs the necessary work. This function usually replies to the initiator, but it may forward the message to another context instead. For this context, the dialog is now over.

2. A **PsmContext** supports a stateful dialog. When the initial message arrives at its destination, it creates two objects:
   - a PSM that receives and sends all messages in the dialog, and
   - a *PsmContext* that owns the PSM.

3. An **SsmContext** also supports a stateful dialog. In this case, the initial message creates three objects:
   - a PSM through which all incoming and outgoing messages in the dialog are passed;
   - an SSM that interacts with the PSM and that also creates one or more additional PSMs in the course of performing the work associated with the original dialog;
   - an *SsmContext* that owns the PSMs and the SSM.

A *PsmContext* or *SsmContext* replies to the initiator, and its objects remain allocated so that they can process subsequent messages.

A context also has a message queue. When an incoming message arrives, it is placed in this queue, and the context is then placed on a work queue. Because a *MsgContext* only supports transaction processing, its queue contains only one message. However, when a *PsmContext* or *SsmContext* is waiting on a work queue, it can receive another message. When this occurs, the new message is appended to the context's message queue.

When a context is removed from a work queue, it passes the incoming message to the appropriate object for processing. A *MsgContext* passes the message to the factory that wrapped the message. A *PsmContext* or *SsmContext* passes the message to the appropriate PSM. If its queue contained more than one message, the context processes each one sequentially.

COMMUNICATION

SessionBase uses three forms of communication:

1. *Function calls* are used between objects running in the *same* context.

   The use of function calls between objects running in *different* contexts is prohibited. This minimizes coupling and provides flexibility in assigning contexts to processors.

2. *Asynchronous messaging* is used between objects running in different contexts. A stateful context sends and receives all of its messages through PSMs.

3. *Synchronous messaging* is supported between two contexts running in the *same* processor on behalf of the *same* user. This eliminates race conditions when a group of contexts is used to reduce coupling. The messages between these contexts are sent as if they were asynchronous, but using a high priority which ensures that messages from outside the group will not be processed until the work within the group has been completed.

   This form of messaging is not supported between contexts running in *different* processors, as it would significantly increase latency. Neither is it supported between contexts running on behalf of different *users*, because a different processor should be able to serve each user.

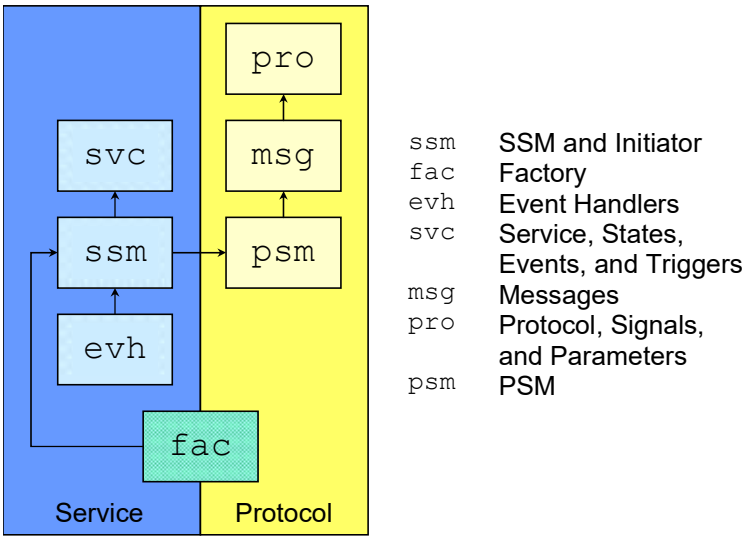
DESIGNING AN APPLICATION

This section is an overview of how to assemble a SessionBase application. A detailed discussion of base classes is left for later. This section only describes the subclasses that an application needs to provide and how to organize them.

SessionBase is used to implement services and the protocols that drive them. Although an application designer invariably implements a service, the protocols that the service uses might already have been implemented.

Protocols and services are implemented separately because each end of a protocol terminates on a service. Say that services S_1 and S_2 communicate using protocol P. If S_1 and S_2 will run in separate processors, then the three components should be implemented separately so that P and S_1 can be built into one load and P and S_2 can be built into another. This is one of the principles that underlies the recommended module structure[1] for SessionBase applications, which is shown in Figure 4.

Figure 4. Module Structure for Applications.
An arrow denotes an #include relationship. Transitive #includes have been removed.



IMPLEMENTING A SERVICE

To implement a service, the following subclasses must be provided:

- *Service*. One subclass for the service as a whole.
- *States*: One subclass for each of the service's states.

[1] In this section, a *module* means a .h and a .cpp file that implement a class or a set of closely related classes.

- *Events*: One subclass for each of the service's events.
- *Event Handlers*: One subclass for each of the service's event handlers. Each event handler processes a specific event, but sometimes on behalf of more than one state.
- *SSM*: One subclass for the service as a whole.

The service, states, and events are simple subclasses and can therefore be placed in one module. The SSM is a heavier class, so it usually has a module of its own. This module will *#include* the one(s) containing the service, states, and events. Finally, all of the event handlers can be placed in a module that *#includes* the other ones.

IMPLEMENTING A PROTOCOL

To implement a protocol, the following subclasses must be provided.

- *Protocol*. One subclass for the protocol as a whole.
- *Signals*. One subclass for each of the protocol's signals.
- *Parameters*. One subclass for each of the protocol's parameters.
- *Messages*. These are subclassed by protocol and sometimes by role (initiator or recipient) and/or signal.
- *PSMs*. Two subclasses: one for the initiator role and one for the recipient role.

The protocol, signals, and parameters are simple subclasses and can therefore be placed in one module. Messages are heavier classes, often with depth of inheritance, so they should have their own module. If they are subclassed by role, the two branches of the inheritance tree should have their own module. PSMs are also heavy classes, so each should have its own module. A PSM module *#include*s a message module, and a message module *#include*s the protocol module.

IMPLEMENTING A MODIFIER SERVICE

There are two major aspects to implementing a modifier service:

- The service to be observed (modified) must provide one or more *Trigger* subclasses.
- The service doing the observing (modifying) must provide one or more *Initiator* subclasses.

In most other respects, the parent (observed) and modifier (observer) are standard services. Further details on how a modifier's SSM interacts with its parent's SSM are described in a subsequent section.

TRANSACTIONS

This section describes how an incoming message reaches the SessionBase application objects that perform the work associated with the message.

I/O THREADS AND INPUT HANDLERS

SessionBase uses the I/O capabilities of NetworkBase: an **I/O thread** reads a byte stream from a socket and passes it to a protocol-specific **input handler**. The input handler for SessionBase applications creates an **SbIpBuffer** to wrap the physical message. This is a simple object that contains the byte stream, as well as the source and destination IP addresses.

In order to be delivered to a SessionBase application, a message requires a SessionBase header. When a message arrives from an internal source, this header is already present. But when the source is external, the input handler must prepend the header.

One of the header's fields identifies the factory that will receive the message, which allows *Factory.AllocIcMsg* to be invoked. This application-specific function creates a message object that provides higher level functions for reading the *SbIpBuffer's* byte stream. After the message has been created, *Factory.ReceiveMsg* is invoked. This function is polymorphic, based on the type of context that the factory uses:

- *MsgFactory.ReceiveMsg* creates a *MsgContext*.

- *PsmFactory.ReceiveMsg* looks at the message header, which also contains a field that identifies the PSM which will receive the message. If this field specifies a PSM, the PSM's context is found. An initial message, however, creates a PSM. This will occur later; for now, the factory just creates a *PsmContext*.

- *SsmFactory.ReceiveMsg* does the same thing as *PsmFactory.ReceiveMsg* if the header specifies a PSM. To handle an initial message, the factory creates an *SsmContext*.

Now that the destination context has been created or found, the message is put on the context's message queue, and the context is put on a work queue that belongs to an **invoker pool**.

INVOKER THREAD

SessionBase objects run on an **invoker thread**. An invoker pool may own a pool of invoker threads, but they are homogeneous and know nothing about applications. An invoker simply enters a loop in which each iteration is a **transaction**. A transaction consists of

- dequeuing a context from a work queue,
- dequeuing a message from the context,
- invoking application software to perform the work associated with message, and
- sending any outgoing messages that the application created.

The invoker calls *Context.ReceiveMsg* to handle a single transaction. Like *Factory.ReceiveMsg*, this function is also polymorphic, based on the type of context:

- *MsgContext.ReceiveMsg* invokes the polymorphic *Factory.ProcessIcMsg* so that the factory specified as the message's destination can process it.

- *PsmContext.ReceiveMsg* invokes *ProtocolSM.ReceiveMsg* on the PSM that received the message. If the PSM does not yet exist, it first invokes *Factory.AllocIcPsm* to create it.

- *SsmContext.ReceiveMsg* also invokes *ProtocolSM.ReceiveMsg* on the PSM that received the message (again, an initial message creates the PSM). The PSM usually returns an event that is passed to an SSM by invoking *ServiceSM.ProcessEvent*. If the SSM does not yet exist, *Factory.AllocRoot* is invoked to create it. This SSM is called the **root SSM** because it provides the context's fundamental behavior.

Each SessionBase transaction runs to completion. In other words, application software *always* returns to the invoker. The invoker cannot be preempted by other application software, and the use of synchronous RPCs or other blocking operations by applications is prohibited. Running to completion

- reduces scheduling costs and latencies,
- eliminates deadlock and priority inversion issues,
- avoids obscure bugs by eliminating semaphores within applications, and
- ensures that applications implement proper state machines.

The invoker tracks the running context, which allows the "running objects" to be accessed as follows:

- *Context.RunningContext* returns the running context.
- *Context.ContextMsg* returns the incoming message that is being processed.
- *Context.ContextPsm* returns the PSM (if any) that received the incoming message.
- *Context.ContextRoot* returns the SSM (if any) that is associated with the running context.

A C++ exception is defined so that a context can commit suicide when an unrecoverable error occurs.[2] Whether an invoker traps or application software raises a fatal exception, recovery proceeds in the same way. The invoker logs the objects in the context for debugging purposes and then invokes their *Cleanup* functions. A *Cleanup* function is similar to a destructor. However, it only deletes private resources, because the invoker recovers the context's SessionBase objects. Only the running context gets cleaned up—the invoker and all other contexts survive. After the context is cleaned up, the invoker goes on to the next transaction.

[2] An unrecoverable error is a serious error, such as the occurrence of a state-event pair for which a service has failed to provide an event handler.

TIMER THREAD

Given that application software is not allowed to use synchronous RPCs or other blocking operations, all timeouts occur when an expected message fails to arrive within a specified interval. SessionBase therefore implements a timeout by sending a message to a PSM. In fact, the only way that an SSM can run is to receive an event from a PSM which, in turn, has received some message. To support timers, the base class PSM provides the functions *StartTimer* and *StopTimer*.

Because applications use timers frequently, SessionBase needs a lightweight timer mechanism. A thread-based timer cannot be used because thousands of SessionBase contexts all run on a small set of invoker threads.

SessionBase implements lightweight timers using a timewheel, which is a type of circular buffer. Each slot in the timewheel represents a future point in time, and the timers that will expire at that time are queued against that slot. The PSM function *StartTimer* (*StopTimer*) causes a timer request to be enqueued against (dequeued from) the appropriate slot on the timewheel. The timewheel is served by a **timer thread** that regularly wakes up, advances to the next slot, and sends messages to all of the PSMs whose timers have expired.

CLASSES

This section discusses SessionBase classes in greater detail, providing information about

- how applications subclass the framework classes;
- when and how objects derived from framework classes are created and destroyed;
- functions that are particularly relevant to applications.

FACTORY

***Purpose**:* provides a concrete factory that creates messages, PSM, and SSMs
***Subclasses**:* one for each type of protocol; further subclassed according to role (initiator or recipient)
Cardinality: singleton
***Construction**:* during system initialization
Destruction: during reload restarts
***Accessed**:* by *Factory_Id*, through a global registry of all factories

The base class *Factory* is an abstract factory that defines functions for creating SessionBase objects. Each concrete factory is a singleton that implements these functions. A concrete factory supports either the initiator or recipient role in a dialog, which means that factory subclasses are based on protocol and role.

A factory is the first application object to be invoked when a message is received. A factory has an identifier that corresponds to its class, and the identifier of the destination factory is included in each message header. This allows the following functions to be invoked on that factory:

- *Factory.AllocIcMsg* creates the appropriate subclass of message object to wrap the incoming physical message.
- In a *MsgContext*, *Factory.ProcessIcMsg* handles the incoming message.
- In a *PsmContext* or *SsmContext*, *Factory.AllocIcPsm* creates the PSM that will process an initial message.
- In an *SsmContext, Factory.AllocRoot* creates the SSM that will process an initial message.

CONTEXT

***Purpose**:* associating the objects that are handling a dialog
***Subclasses**: MsgContext* (connectionless protocols); *PsmContext* and *SsmContext* (connection-oriented protocols)
***Cardinality**:* two per dialog (at initiator and recipient)
***Construction**:* created by an initial message
***Destruction**:* destroyed by the framework if the context has no PSMs at the end of a transaction
***Accessed**:* through *Context.RunningContext*; PSMs and SSMs also have functions that return their context

A context associates the objects that are handling a dialog. It is created when an initial message arrives. See “Contexts” on page 10.

When an invoker dequeues a message from a context's message queue, it invokes the polymorphic function *Context.ProcessIcMsg* to handle the message:

- *MsgContext.ProcessIcMsg* invokes *Factory.ProcessIcMsg* to handle the message. After the message has been handled, the context destroys itself, which ends the dialog.

- *PsmContext.ProcessIcMsg* invokes *ProcessIcMsg* on its PSM. After the message is handled, the context sends any outgoing messages queued on the PSM. If the PSM is now in the idle state, the context destroys the PSM and itself, ending the dialog.

- *SsmContext.ProcessIcMsg* also invokes *ProcessIcMsg* on the PSM that received the message. After the message has been handled, the context sends any outgoing messages queued on its PSMs and destroys any PSMs in the idle state. If no PSMs remain, the SSM should be in the null state, in which case the context destroys the SSM and itself, ending the dialog.[3]

When an irrecoverable error occurs, SessionBase invokes *Context.Kill* to raise an exception, which logs the context and destroys it. Applications may also use this function.

PROTOCOL STATE MACHINE (PSM)

***Purpose**:* contains run-time data for the usage of a protocol
***Subclasses**:* one for each type of protocol; further subclassed by role (initiator or recipient)
***Cardinality**:* one for each run-time instance of a protocol role
***Construction**:* created by an initial incoming message (if the recipient) or just prior to building an initial outgoing message (if the initiator)
***Destruction**:* destroyed by the framework if in the idle state at the end of a transaction
***Accessed**:* typically through an SSM-specific function; also through the iterator functions *Context.FirstPsm* and *Context.NextPsm*

A PSM implements the state machine associated with either the initiator or recipient role in a stateful dialog. PSM subclasses are therefore based on protocol and role. A PSM sends and receives asynchronous messages. A dialog that is stateful at both ends is implemented by two PSMs, in which case each PSM has a **peer** PSM.

A factory accessed through the factory registry creates the PSM that will receive an initial message. A PSM that will send an initial message is created by an SSM, which explicitly specifies the PSM's class.

Each PSM is part of a **protocol stack**. At the bottom of each stack is a **port** (**MsgPort**), a non-virtual framework class that serves as the stack's I/O address. If a stack has a peer stack that has sent it a message, the port also knows the peer stack's I/O address. The addresses of the source and destination ports are included in each

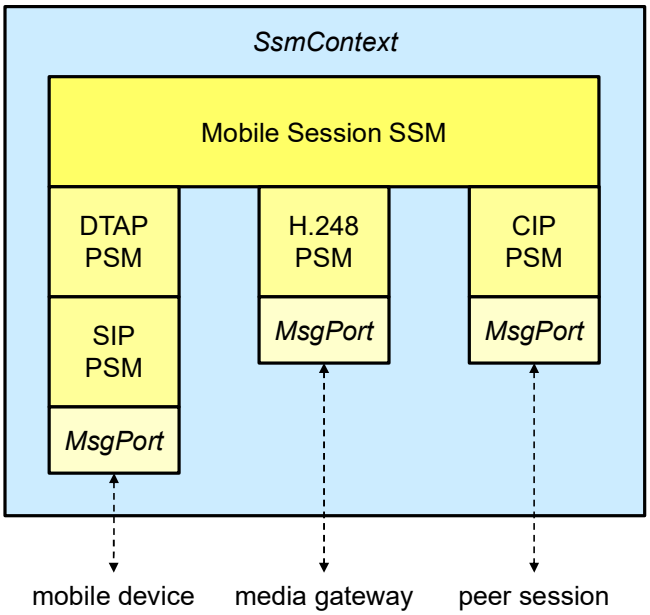
[3] It is an error if (a) the root SSM is in the null state, but not all PSMs are in the idle state; (b) the root SSM is not in the null state, but all PSMs are in the idle state. Either error causes an exception and the cleanup of the context.

SessionBase message header. A port's address is actually the index of its object block in the port object pool. This allows the destination port to be quickly located when a message arrives.

A stack usually has a single PSM above its port. However, PSMs can be stacked. If a stack has more than one PSM, each PSM unwraps (decapsulates) an incoming message as it travels up the stack and wraps (encapsulates) an outgoing message as it travels down the stack. The PSMs in the stack must implement the **ProtocolLayer** virtual functions *AllocUpper*, *AllocLower*, *WrapMsg*, and *UnwrapMsg*. *ProtocolLayer* is the base class for both ports and PSMs, so a port also implements these functions. But it does so in a trivial way, because messages between a port and the lowermost PSM are neither encapsulated nor decapsulated.

In an SSM context, the root SSM is effectively upper layer of each protocol stack. In Figure 5, a mobile call SSM is using DTAP over SIP to communicate with a user's mobile device, CIP to communicate with the peer user's session, and H.248 to communicate with a media gateway.

Figure 5. An SSM and its Protocol Stacks



The base class PSM also provides the functions *StartTimer* and *StopTimer*. When a timer expires, the PSM receives a timeout message that contains the timer identifier that was originally provided to *StartTimer*. *StartTimer* is a PSM function because a stateful context can only receive a message, including a timeout, through a PSM.

INCOMING MESSAGES

Recall that, when a message arrives at I/O level, an input handler places it in an *SbIpBuffer*, adding a SessionBase header if necessary. A factory then allocates a message to wrap the buffer. At this point, the destination context must be found so that the message can be queued on it. This is done by looking at the destination port address in the message header. If the port exists, the message is queued against the port's context. An initial message,

however, only specifies the destination factory. In this case, the factory must create a context and port to receive the message. It then updates the message header to reference the new port.

When a context is removed from a work queue to process a message, it invokes *ReceiveMsg* on the port on which the message arrived. The port, in turn, invokes *ReceiveMsg* on the layer above it, which is always a PSM. If the PSM does not exist, the port creates it by invoking *AllocIcPsm* on the destination factory in the message header.

The framework implements *ReceiveMsg* for a PSM. It simply adds the message to the PSM's incoming message queue and invokes *ProcessIcMsg*, a virtual function that implements the incoming side of a PSM's state machine. This function ultimately does one of three things:

- Returns true after allocating an event for the root SSM. Unless the PSM has detected a protocol error, this is always the Analyze Message event. The event is then passed to the root SSM. If the SSM does not yet exist, *Factory.AllocRoot* is invoked to create it.

- Returns true without allocating an event. This means that the message should continue up the stack. It is decapsulated by invoking *UnwrapMsg*, followed by *ReceiveMsg*, on the upper PSM. If an upper PSM does not yet exist, *AllocUpper* is invoked on the current PSM to create it.

- Returns false, which ends the transaction. This is unusual, because the PSM handled the message by itself. A SIP PSM might do this with a 100 TRYING, for example.

OUTGOING MESSAGES

When a protocol stack will send an initial message, an SSM creates the uppermost PSM. It then creates a message, which is queued on the PSM. Indeed, each outgoing message is queued against the uppermost PSM in a protocol stack.

Messages are not sent until the end of the transaction. The context maintains a queue of PSMs, which are ordered from uppermost to lowermost in their respective stacks. It traverses this queue and invokes *ProcessOgMsg* on each PSM that has a pending outgoing message. This function implements the outgoing side of the PSM's state machine, and it ultimately does one of three things:

- Returns *SendMessage*, which causes the message to be passed to the layer below. If that layer does not exist, the PSM's *AllocLower* function is invoked to create it. The default implementation of *AllocLower* creates a port. *WrapMsg* is then invoked on the layer below, followed by *SendMsg*. If the lower layer is a PSM, *SendMsg* adds the message to the PSM's outgoing message queue; the context will encounter the PSM soon thereafter, as it continues traversing the PSM queue. If the lower layer is a port, it invokes *Message.Send* to actually send the message.

- Returns *PurgeMessage* to delete the message. This is rare but might occur when bundling messages.

- Returns *SkipMessage* to continue with the next message. This is also rare.

Each outgoing message in a protocol stack is queued on the PSM below, rather than being immediately passed all the way down the stack and sent. This allows a PSM to bundle messages. Before *ProcessOgMsg* is invoked for each message, *PrepareOgMsgq* is invoked so that the PSM can bundle messages in the queue or even create a message that it needs to send.

When an application builds an initial message, it must include the source and destination IP addresses and IP ports in the message header. For an intrasystem message, it must also include the destination factory. The port (at the bottom of the stack) inserts its I/O address and the source factory (of the PSM above) before it sends the message.

When an application builds a subsequent message, it does not have to include the addresses. The port inserts them before sending the message.

After the PSM queue has been traversed, any PSMs in the idle state are destroyed. A context is destroyed when it no longer has any PSMs.

SERVICE STATE MACHINE (SSM)

***Purpose**:* contains run-time data for the usage of a service
***Subclasses**:* one for each type of service
***Cardinality**:* one for each run-time instance of a service
***Construction**:* created by an initial message
***Destruction**:* destroyed by the framework if in the null state at the end of a transaction
***Accessed**:* by *Context.ContextPsm.RootSsm* (for a root SSM)

An SSM contains per-context data for a state machine that coordinates the behavior of multiple PSMs. An SSM is therefore subclassed to implement an application's state machine.

Factory.AllocRoot creates the root SSM when an initial message arrives. An *SsmContext* destroys the root SSM after it has destroyed the last PSM.

SessionBase provides observer patterns that allow an SSM to collaborate with other SSMs that run in the same context. This allows applications to interact while remaining loosely coupled, with supplementary service logic remaining separated from basic service logic. The details are discussed in “Initiator” on page 28.

An SSM usually coordinates the behavior of two or three PSMs. If a context requires additional PSMs, other SSMs that interact with the root SSM using the above-mentioned observer patterns usually create them. This helps to preserve loose coupling and often results in software reuse.

MESSAGE

***Purpose**:* provides a wrapper for a physical message
***Subclasses**:* by protocol, role (initiator or recipient), and sometimes signal
***Cardinality**:* one for each physical message that exists at run-time
***Construction**:* when a message arrives (incoming) or needs to be built and sent (outgoing)

***Destruction**:* after a message has been processed (incoming) or sent (outgoing), unless the message's *Save* function has been invoked, in which case it is not destroyed until the *Unsave* function drops the save counter to zero
***Accessed**:* through *Context.ContextMsg* (the incoming message being processed) and various PSM functions (for saved messages)

A message provides a wrapper for a buffer (physical message). Applications subclass from various base message classes to provide protocol-specific or even signal-specific behavior.

INCOMING MESSAGES

An incoming message is created when the input handler invokes *AllocIcMsg* on a factory accessed through the factory registry. What happens next depends on whether the message is destined for a stateless or stateful context.

If the message is for a stateless context, the input handler queues the message on a new *MsgContext*, which it then adds to a work queue. When the invoker dequeues the context, it invokes the *ProcessIcMsg* function on the destination factory, which is specified in the message header. The invoker then destroys the message.

If the message is for a stateful context, the input handler queues it on context that owns the PSM that will receive the message. The input handler then adds the context to a work queue. When the invoker dequeues the context and invokes its *ProcessIcMsg* function, the context moves the message to the destination PSM's **incoming message queue**. This queue is LIFO so that the most recent message is always at the front.

The context destroys the message at the end of the transaction unless an application invoked the message's *Save* function. A saved incoming message remains on the PSM's LIFO incoming message queue. Saving an incoming message is useful if, for example, it contains information that may be needed during a subsequent transaction.

OUTGOING MESSAGES

In a stateless context, a factory creates any outgoing messages. The factory then sends the message, at which point the message is destroyed.

In a stateful context, an event handler or SSM (or, occasionally, a PSM) creates outgoing messages. Each outgoing message is placed on the FIFO **pending message queue** of the PSM through which it will be sent. At the end of the transaction, the context iterates over its PSMs to send all pending messages. A message is only sent at the end of the transaction so that other event handlers and applications can alter its contents before it is sent.

After an outgoing message is sent, it is destroyed unless an application invoked the message's *Save* function. A saved outgoing message is placed on its PSM's LIFO **outgoing message queue**. Saving an outgoing message is useful if, for example, it must be retransmitted if a timer expires.[4]

[4] An application should only retransmit a message that was sent on an external interface.

PROTOCOL

***Purpose**:* provides a registry for a protocol's signals and parameters
***Subclasses**:* one for each type of protocol
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *Protocol_Id*, through a global registry of all protocols

A protocol consists of a set of signals (message types) and parameters. It also specifies the order in which its signals may be sent, although the implementation of resides within the PSMs that implement the protocol.

Each protocol subclass is a flyweight that supports some protocol. It has two registries, one for its signals and one for its parameters. The protocol associated with a physical message is specified by an identifier that is included in the SessionBase message header. This allows the protocol's flyweight to be found in a global protocol registry.

Messages, PSMs, and SSMs are primarily subclassed by protocol and role, so why not delegate the creation of these objects to protocols instead of factories? The reason is that delegating to a factory allows more than one message, PSM, or SSM subclass to support a particular protocol role. A factory creates the subclasses that are appropriate to a particular *application*. Consider a pipes-and-filters pattern in which a series of state machines uses the same underlying protocol. Although the SSMs that implement the state machines might use the same PSMs, each SSM would be subclassed according to its filtering algorithm—hence the need to allow subclassing by more than protocol and role.

SIGNAL

***Purpose**:* provides information about a signal within a protocol
***Subclasses**:* one for each of a protocol's signals
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *Signal_Id*, through its protocol's signal registry

A signal is a message type in a protocol. The signal associated with a physical message is specified by an identifier that is included in the SessionBase message header. This allows the signal's flyweight to be found in its protocol's signal registry.

PARAMETER

***Purpose**:* provides information about a parameter within a protocol
***Subclasses**:* one for each of a protocol's parameters
***Cardinality**:* singleton
***Construction**:* during system initialization

***Destruction**:* during reload restarts
***Accessed**:* by *Parameter_Id*, through its protocol's parameter registry

A parameter contains one or more fields that provide additional information associated with a signal in a protocol. In SessionBase, a parameter subclass is a flyweight that provides symbolic display capabilities and that specifies

- whether the parameter is of fixed or variable length, and
- whether the parameter is mandatory, optional, or illegal for each signal.

Within a physical message, a parameter is encoded using a **type-length-value** (**TLV**) layout. The format of a physical message (ignoring headers and any encapsulation) is therefore *PS(TLV)**, where

- *P* is the protocol identifier,
- *S* is the signal identifier,
- *T* is a parameter identifier,
- *L* is a parameter length, and
- *V* is a parameter *struct*.[5]

A parameter flyweight is accessed through its protocol's parameter registry. Because it is a flyweight, its instance data must be passed as an argument to any function that needs it. For example, its *DisplayMsg* function is passed a byte stream that is embedded in the physical message being displayed.

If a new software release appends fields to a parameter's *struct*, the *PS(TLV)** encoding ensures that previous releases can still parse the parameter. The encoding also avoids passing objects between processors, which is costly and messy. Much is gained and little is lost by wrapping a physical message with a message object and using the *PS(TLV)** encoding to access protocol, signal, and parameter flyweights.

SERVICE

***Purpose**:* provides a registry for a service's states and event handlers
***Subclasses**:* one for each service
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *Service_Id*, through a global registry of all services

A service provides registries for the flyweights that implement an application's state machine. The instance data for a service's state machine resides in an SSM, but all of the SSMs that are running the same application share a singleton service. Each service has an identifier that it uses to place itself in the global service registry.

[5] This is the recommended format for parameters. However, not all protocols defined in standards use this format, in which case their message and parameter classes will vary somewhat from the descriptions in this document.

EVENT

***Purpose**:* defines an event that can be handled by a state machine
***Subclasses**:* one for each of a service's events
***Cardinality**:* usually a singleton, because the running SSM handles one event at a time
***Construction**:* created by a PSM (the first event in each transaction), a message analyzer, or an event handler
***Destruction**:* destroyed by the framework after being processed by an event handler
***Accessed**:* passed as an argument to event handlers

An event is an input that is handled by an SSM. Each event subclass has its own event identifier.

Events are internal to a state machine and are therefore distinguished from signals. When a PSM receives a message, its protocol state determines if the message is legal. The PSM then raises either the Analyze Message event or a protocol error event. The Analyze Message event is handled by a **message analyzer**, which is a special type of event handler. A message analyzer looks at the message (primarily its signal) and the SSM's state, which allow it to raise an appropriate application-specific event. It is this latter event that actually causes one of the SSM's event handlers to take action.

In many cases, a message analyzer maps a signal directly to an event. However, there are protocols in which signal S_1 causes the set of actions A_1 to be performed, signal S_2 causes the set of actions A_2 to be performed, and $A_1 \cap A_2 \neq \emptyset$. In situations like this, it is often desirable to decompose A_1 and A_2 into a sequence of events, which is one reason for distinguishing events from signals. Another reason for the distinction is that there are protocols in which the meaning of a signal depends on an application's state. Soft key events (e.g. F1, F2) are a good example. These situations are best handled by also defining events that specify the actual work to be performed, so as to decouple the interface (e.g. soft keys, control characters, or menu selections) from the implementation.

STATE

***Purpose**:* provides a registry for the event handlers that support a state
***Subclasses**:* one for each of a service's states
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *State_Id*, through its service's state registry

A state is a flyweight that defines a state in a service's state machine. It contains an array that specifies the event handler to invoke for any event that occurs in its state. Each state has an identifier that it uses to place itself in a state registry owned by its service.

EVENT HANDLER

***Purpose**:* handles a state-event pair that can occur in a state machine
***Subclasses**:* one for each of an SSM's event handlers

***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *EventHandler_Id*, through its service's event handler registry

An event handler is a flyweight that implements the work associated with the arrival of some event in some state. Each event handler has an identifier that it uses to place itself in an event handler registry owned by its service. This identifier, rather than a pointer to the event handler, is also placed in each state's event handler registry, which is indexed by the identifier of the event to be handled.

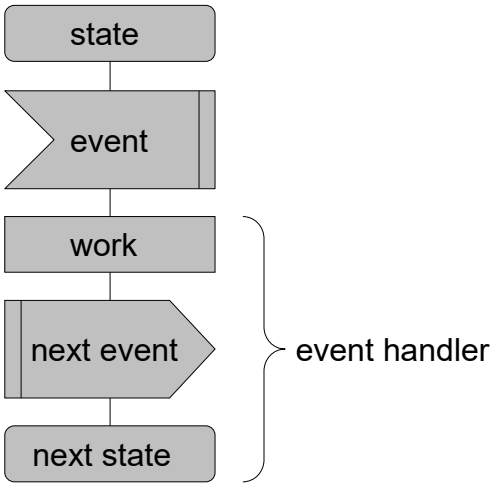
An event handler is a function that is wrapped by an object for reuse (inheritance) purposes. It performs application-specific work, which may include creating outgoing messages that are queued on PSMs and sent at the end of the transaction. Before it returns, the event handler may also

- set the next state by invoking *SetNextState* on its SSM, and/or
- create the next event to be processed, so that another event handler can perform additional work.

The object model for states, event, and event handlers is based on SDL (Specification and Description Language) diagrams, which telecom standards often use to describe the detailed service behavior. A simple SDL diagram appears in Figure 6, which describes the following:

- An event arrives in the current state.
- An event handler performs some work.
- The event handler raises the next event.
- The event handler updates the state.

Figure 6. SDL Diagram for an Event Handler



The SessionBase implementation for this is that each SSM contains a reference to its service, its current state, and the event that it is processing. The appropriate event handler is then invoked as follows:

```
ehid = this->service->states[this->currState->id][this->currEvent->id];
this->nextEvent = this->service->eventHandlers[ehid]->handle(this);
```

Conceptually, an event handler is an SSM function. But it is implemented as a flyweight, so it must receive its SSM as an argument, and it must use *get* and *set* functions to read and write SSM data. In practice this rarely turns out to be much of a penalty. Where performance is critical, it can be improved by inlining high-runner *get* functions.

Why *not* make event handlers functions on SSMs? This is a common practice in object-oriented state machines, in which event handler functions are invoked using a technique called *double dispatching*. However, there are many advantages to implementing the event handlers as singletons:

- Indexing by state and event identifiers to select an event handler is intuitive.
- An object browser can easily display the event handler associated with a given state and event.
- The decoupling of states, events, and event handlers allows state machines to be modified with less recompilation.
- It is easy to modify—or even add—a state or event handler and register it *in a running system*, which helps to achieve continuous availability.
- Each event handler can be inherited or subclassed independently.
- The framework performs all event dispatching in a *ProcessEvent* function defined by the base class SSM. This makes it impossible to add inappropriate code to event dispatching software.[6] All state machine logic resides within event handlers. And because the framework does the dispatching, it is possible to provide observer patterns that are available to, and consistent among, all state machines.

TRIGGER

***Purpose**:* provides a registry for Initiators
***Subclasses**:* one for each of a service's triggers
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by *Trigger_Id*, through its service's trigger registry

A trigger represents one or more state-event pairs whose occurrence can be monitored to trigger the creation of another SSM in the same context. Each trigger is a singleton and has an identifier that it uses to place itself in a trigger registry owned by its service. If an SSM supports initiators, its event handlers invoke two functions to specify the triggers associated with a state transition:

[6] An example of such software is, "If the conditions under which customer report so-and-so occurs are true, then override the next event or state set by the previous event handler." This is state machine logic and, as such, belongs in an event handler, not in the event dispatcher. Such defilement of the event dispatching software is almost certain in the absence of a formal state machine framework. The importance of having the framework preclude this kind of hacking cannot be overstated.

- *SetNextSnp* indicates what the service just did, which allows observers to *augment* the SSM's behavior. The trigger identifier accounts for the current state, the event that was just handled, and the next state.
- *SetNextSap* indicates what the service plans to do next, which allows observers to *override* the SSM's behavior. The trigger identifier accounts for the next state and the next event.

INITIATOR

***Purpose**:* allows a modifier service to observe the behavior of its parent's SSMs to create its own SSM
***Subclasses**:* one for each of the triggers that are of interest to a modifier service
***Cardinality**:* singleton
***Construction**:* during system initialization
***Destruction**:* during reload restarts
***Accessed**:* by traversing a queue that belongs to a trigger

An application that wishes to observe the behavior of another SSM, in order to trigger the creation of its own SSM in the same context, implements one or more initiators. Each initiator registers with a trigger that is owned by the service of the SSM whose behavior the initiator wishes to observe. A trigger maintains a queue of the initiators that have registered with it.

When an event handler has invoked *SetNextSap* or *SetNextSnp* to indicate that a trigger has occurred, the initiator is given the opportunity to create its SSM. An SSM created in this manner is called a **modifier**, because its purpose is to modify the behavior of its **parent**.

A modifier refactors optional behavior out of its parent, which keeps the parent's state machine simpler. Because a modifier collaborates with its parent by observing the parent's states and events, it can run atop all parents that share a common state-event space. In a call server, for example, each supplementary service is implemented as a modifier of a basic call parent.

MODIFIER SSMS

An initiator requests the creation of its SSM by raising the Initiation Request event. Any modifier SSM that already exists can react to this event by denying initiation of the modifier if the two modifiers are incompatible. Whether or not this occurs, the new modifier SSM is created.

SessionBase defines five event handlers as functions on a modifer SSM:

- *ProcessInitAck* is invoked after creating the SSM if no modifier denied its initiation.
- *ProcessInitNack* is invoked after creating the SSM if another modifier denied its initiation.
- *ProcessSip* (service initiation point) is invoked to allow the SSM to deny the initiation of another modifier.
- *ProcessSap* (service alteration point) is invoked *before* the parent SSM handles an event. It allows the modifier to override the parent's behavior by reacting to the event itself.
- *ProcessSnp* (service notification point) is invoked *after* the parent SSM handles an event. It allows the modifier to augment the parent's behavior by performing additional actions.

EVENT ROUTING INSTRUCTIONS

If SessionBase did not support modifiers, things would be simple. If an event handler did not raise another event, the transaction would end. If it raised another event, the next event handler would be invoked. The inclusion of modifiers, however, creates the need for collaboration between a parent and its modifiers. This is supported by having event handlers return one of the following event routing instructions:

- *Suspend* by ending the transaction when the work associated with the incoming message is complete.
- *Continue* after raising a new event that will be handled by the same SSM.
- *Pass* the current event onward. The parent will handle the event unless another modifier intervenes.
- *Revert* to the parent after raising a new event, usually after the modifier has performed some work itself.
- *Initiate* a modifier. A modifier SSM is created either by an initiator, as previously described, or by its parent, when reacting to a message that contains an initiation request.
- *Resume* processing a pending event that was saved in order to send a request and receive a reply that would determine how to continue.

SUMMARY OF CLASSES

Table 1 summarizes the classes that were discussed in this section and how they are subclassed to construct applications.

Over half the base classes defined by SessionBase rely solely on singleton subclasses. Only six (contexts, messages, PSMs, timers, SSMs, and events) create objects at run time. Table 2 summarizes the singletons and registries used by SessionBase.

Table 1. Classes and Subclassing Strategies

Class	Purpose	Subclassing Strategy
Context	associates the objects that implement a dialog	*MsgContext*, *PsmContext*, *SsmContext* (all framework subclasses)
Event	provides an input to an SSM	by each event required by a service
Event Handler	handles a state-event pair that can arise in a service	by state-event pairs that require different event handlers
Factory	provides a concrete factory that creates messages, PSMs, and SSMs	by protocol and role
Initiator	allows one SSM to be created by observing the behavior of another SSM	by an application whose SSM is created by observing the behavior of another SSM
Message	wraps a physical message	by protocol, role, and possibly signal
Parameter	provides additional information for one or more signals	by each parameter in a protocol
Protocol	provides registries for the signals and parameters that define a protocol	by each protocol
PSM	implements the state machine associated with a role in a protocol	by protocol and role
Service	provides registries for an application's states, event handlers, and triggers	by each application
Signal	defines a message type in a protocol	by each signal in a protocol
SSM	provides per-context data for a service that coordinates the behavior of multiple PSMs	by each application
State	defines a state for a service	by each state required by a service
Trigger	defines a state-event pair whose occurrence may be observed by an initiator	by one or more state-event pairs whose occurrence may be observed by an initiator

Table 2. Singletons and Registries

Subclass of…	Singleton?	Registry for subclass	Scope of registry	Registry indexed by…
Event Handler	yes	*Service.handlers_*	per-service	event handler identifier
Factory	yes	*FactoryRegistry*	global	factory identifier
Initiator	yes	*Trigger.initq_*	per-trigger	none
Parameter	yes	*Protocol.parameters_*	per-protocol	parameter identifier
Protocol	yes	*ProtocolRegistry*	global	protocol identifier
Service	yes	*ServiceRegistry*	global	service identifier
Signal	yes	*Protocol.signals_*	per-protocol	signal identifier
State	yes	*Service.states_*	per-service	state identifier
Trigger	yes	Service.triggers_	per-service	trigger identifier

MESSAGING

This section discusses the SessionBase messaging system, which provides various capabilities that raise the level of abstraction in software that deals with protocols.

PROTOCOLS, SIGNALS, AND PARAMETERS

The software of most systems does a poor job of defining the protocols on which the system is based. Signals are simply *int* constants. Constants for well-known IP ports are the only things that represent protocols. Parameters are *structs* whose relationship to protocols and signals is not defined. Only by closely examining protocol state machines, which are written in an *ad hoc* manner, is it possible to tediously reverse engineer the protocols that underlie the system's behavior. An object model for these concepts—protocols, signals, parameters, and protocol state machines—is desperately needed, and SessionBase provides one.

MESSAGES

In many systems, applications deal with messages at the buffer and byte level. Error prone code for managing buffers and building and parsing byte streams pervades the application software. By wrapping buffers and byte streams with message objects, SessionBase removes such code from applications.

STATEFUL DIALOGS

In many systems, applications send messages to threads, and little more exists in the way of a framework. This is adequate for stateless dialogs. But for stateful dialogs, which are considerably more complex, anything beyond threads and messages becomes *ad hoc*. By providing a full object model for stateful dialogs, SessionBase creates consistency among applications, making it significantly easier for one designer to understand and modify applications developed by others.

ESTABLISHING A DIALOG

When a port sends an initial message, its identifier (the index that its object pool uses to access its object block) is included as the source address in the SessionBase message header. This allows the port to be found quickly when the recipient's reply arrives. Similarly, when a port sends an initial reply, its address is also included in the message header. This way, subsequent messages from the initiator can directly address the port that the initial message created.

To improve performance, an application payload is included in the both the initial message and the initial reply. It is not necessary to establish the association between the source and destination ports before the application can begin to pass data between them.

JOINING A CONTEXT

When a port sends an initial message, the recipient creates a new context when it receives the message. However, there are situations in which the initiator wants to join an existing context. A conference call, for example, can be implemented by a context that creates a PSM for each user who dials into the conference.

The SessionBase message header contains a *join* flag that supports this type of scenario. If an application in the initiator's context sets this flag, *FindContext* is invoked on the recipient's factory, which must be derived from *SsmFactory*. To find the context on which a new port and PSM should be created, a factory looks at application specific data in the message. In the conference call example, the message would contain the directory number that participants used when dialing into the conference. The recipient's factory would find this directory number and use it as a key to look up the context that the participant will join.

CLOSING A DIALOG

When an application sends a final message, it must set the *final* flag in the SessionBase message header. This breaks the association between the two ports so that no further messages can be sent or received. To improve performance, a final message contains application data and is not acknowledged.

The *final* flag ensures that each dialog ends with a final message. If a PSM is destroyed before it has sent a final message, SessionBase invokes its *SendFinalMsg* function, prompting it to send one. This ensures that the peer will receive a final message during the following situations:

- when a PSM enters the idle state but has not sent a final message
- when an SSM enters the idle state before all of the context's PSMs have sent a final message
- when an invoker traps and a context is cleaned up

SessionBase also defines an *InjectFinalMsg* function for PSMs. It is invoked on all affected PSMs when another processor goes out of service. It allows the PSM to build a final message for itself, as if one had arrived from the other processor.

CLOSING A DIALOG BEFORE RECEIVING A REPLY

Assume that

- Port *A* initiates a dialog by sending an initial message that creates port *B*.
- Soon thereafter, *A* wants to send a final message to *B*, closing the dialog.

In some systems, *A* must receive an initial reply before it can close the dialog. This restriction exists because *A* doesn't learn *B*'s address until *A* receives a reply. If this restriction is not removed, it creates artificial complexity for *A*, which must remain allocated for no other reason than to receive *B*'s reply, at which point it can send its final message. SessionBase eliminates this restriction by maintaining, in each processor, a mapping from each initiator's port to the recipient's port. This allows an initiator to close a dialog before it has received a reply.

STALE MESSAGES

Assume that

- A dialog exists between ports *A* and B, and *A* sends message *M* to *B*.
- Before M arrives, *B* sends a final message to *A*, closing the dialog.

When M arrives, it is stale. If *B* has been reassigned to another context, *B* receives *M*, disrupting its work. In many systems, applications must deal with stale messages, which are the source of numerous bugs. SessionBase relieves applications of this burden by using sequence numbers to distinguish a port's incarnations. Each message includes the destination port's sequence number. When the message arrives, SessionBase discards it (as stale) if this sequence number does not match the port's current sequence number.

MESSAGE PRIORITIES

SessionBase defines four message priorities. In order of lowest to highest priority, they are

- *Ingress*, for an initial message to an ingress context.[7]
- *Egress*, for an initial message from an ingress context to an egress context.[8]
- *Progress*, for a subsequent message. If an intraprocessor message is *ingress*, SessionBase upgrades its priority to *progress*.
- *Immediate*, to simulate synchronous messaging as described in "Communication" on page 11.

An application can allow SessionBase to set a message's priority except in the following situations:

- When an input handler constructs the SessionBase header, it must set *progress* priority when appropriate.
- In an outgoing message, the application must set *egress* or *immediate* priority when appropriate.

A context that receives a message is placed on a work queue. An invoker pool has four work queues, one for each message priority. All messages with a higher priority are processed before any messages with a lower priority. When a processor enters overload, input handlers start to discard ingress work. Although non-ingress work is never discarded, the latency (queueing delay) for egress work can increase appreciably during overload.

[7] *Ingress* refers to the first context that is created on behalf of a session *initiator*.

[8] *Egress* refers to the first context that is created on behalf of a session *recipient*. Egress work is distinguished from ingress work because, by the time the egress half of a session is reached, the system has already committed significant resources to handling the ingress half of the session.

SAVING MESSAGES

SessionBase allows incoming and outgoing messages to be saved. How and why this is supported is described in "Message" on page 21.

MESSAGE ENCAPSULATION

The base class **TlvMessage**, which supports the TLV message format, defines the *Wrap* function for encapsulation, as well as a constructor for decapsulation. An encapsulated message resides within a single parameter with the format *ML[PS(TLV)*]*, where *M* is a parameter identifier used by an encapsulated message and the outer *L* is the length of the encapsulated *PS(TLV)** message, which looks like one big parameter to the encapsulating protocol.

MESSAGE HEADER

The SessionBase header (**MsgHeader**) contains the following fields:

- The source and destination addresses (factory identifier, port identifier, and port sequence number).

  In a stateless context, the application must set the source and destination factory identifiers. The port identifier and sequence number are ignored.

  In a stateful context, SessionBase automatically fills in the source address using fields in the source port. Applications only have to set the destination address in an initial message. After that, SessionBase also sets this address because each port knows its peer's address information, which is included whenever a message is sent.

- A priority (see "Message Priorities" on page 34).
- An *initial* flag that is set for an initial message.
- A *final* flag that is set for a final message.
- A *join* flag that is set when an initial message is destined for a context that already exists.
- The message length. This field is set by SessionBase base message classes, which update it in functions such as *TlvMessage.AddParm*.
- The payload, which contains the protocol, signal, and parameters in *PS(TLV)** format. The protocol is set by the PSM through which the message passes. Applications set the signal and provide the parameters.

FOR FURTHER READING

A Pattern Language of Call Processing describes design patterns used in call servers. A call server that supports a large number of supplementary services is a demanding session processing application, so these patterns may be useful in other systems. SessionBase supports most of them. However, there are some terminology differences, so the reader may find the following table helpful:

A Pattern Language of Call Processing	SessionBase
AFE	SSM
Agent	Factory
consume	suspend
feature	service
FSM	Service
PFE	Initiator
PFQ	Trigger
reenter	revert
Transactor	Context