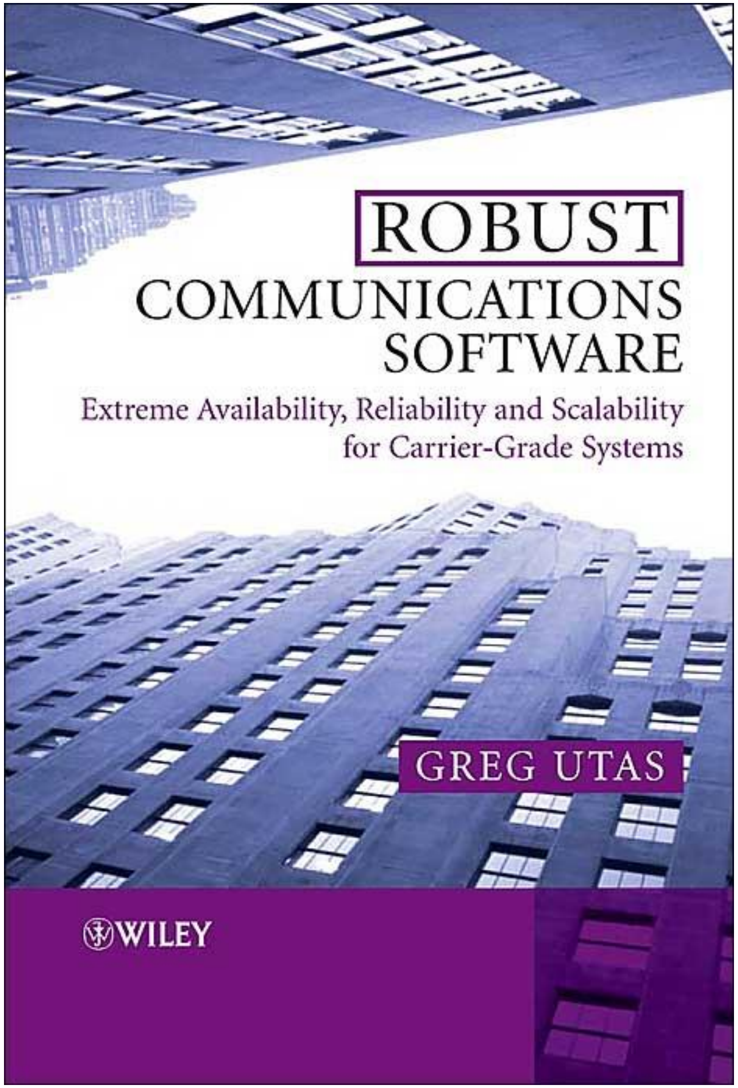
Session Processing

Greg Utas
Pentennea
greg@pentennea.com

Personal Background

- software architecture
- Nortel (20 years)
  - call server frameworks
  - GSM MSC rearchitecture
- Sonim (2 years)
  - push-to-talk for wireless networks
- Pentennea
  - carrier-grade software consulting



Session Processing Patterns

- Definitions
- Protocol
- State Machine
- Separation of Services and I/O
- Separation of Services and Protocols
- Separation of Sessions
- Message Preservation
- Type-Length-Value Message

Definitions

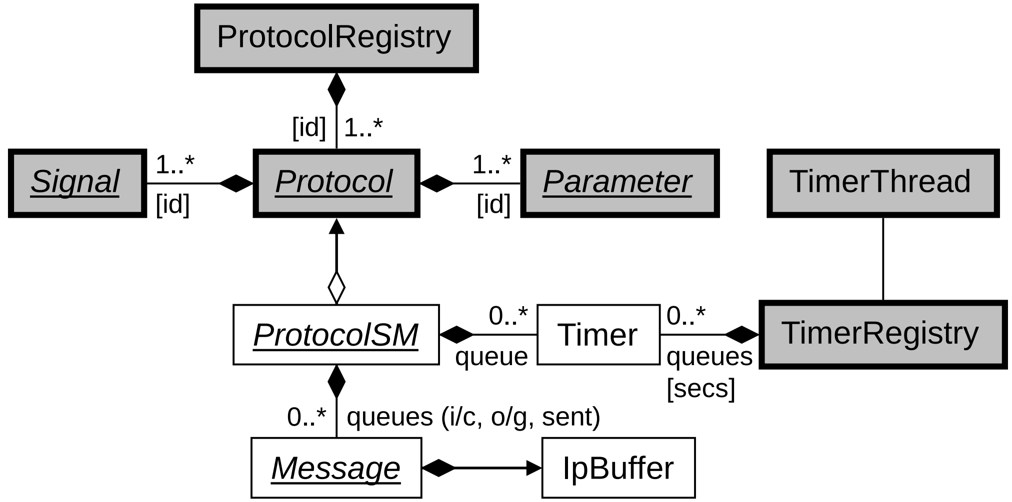
Connectionless protocol	Connection-oriented protocol
stateless	stateful
request-response	setup, data exchange, disconnect
transaction processing	session processing

- transaction: receive message, perform work, send message(s)
  - handling a request in a connectionless protocol
  - handling a message in a connection-oriented protocol
- session: user-network (client-server) dialog
  - based on a connection-oriented protocol
- service: uses protocols to implement a user application
  - may connect users (data exchange phase = user-to-user communication)

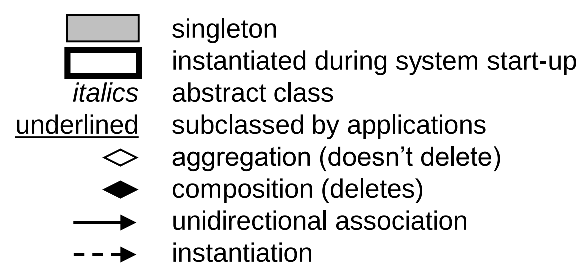
Protocol

- a Protocol defines
  - the order in which Signals (message types) can be sent
  - the Parameters that are mandatory or optional for each signal
- a Message
  - provides functions for parsing and building a byte stream
  - owns an IpBuffer, which contains the byte stream and the source and destination IP addresses
- each message passes through a ProtocolSM (PSM) that
  - implements a state machine to enforce its protocol
  - is created (destroyed) during the transaction in which an initial (final) message in its protocol is sent or received
  - communicates with a conjugate PSM (if an internal protocol) or the outside world (if an external protocol)

Object Model (for Protocols)



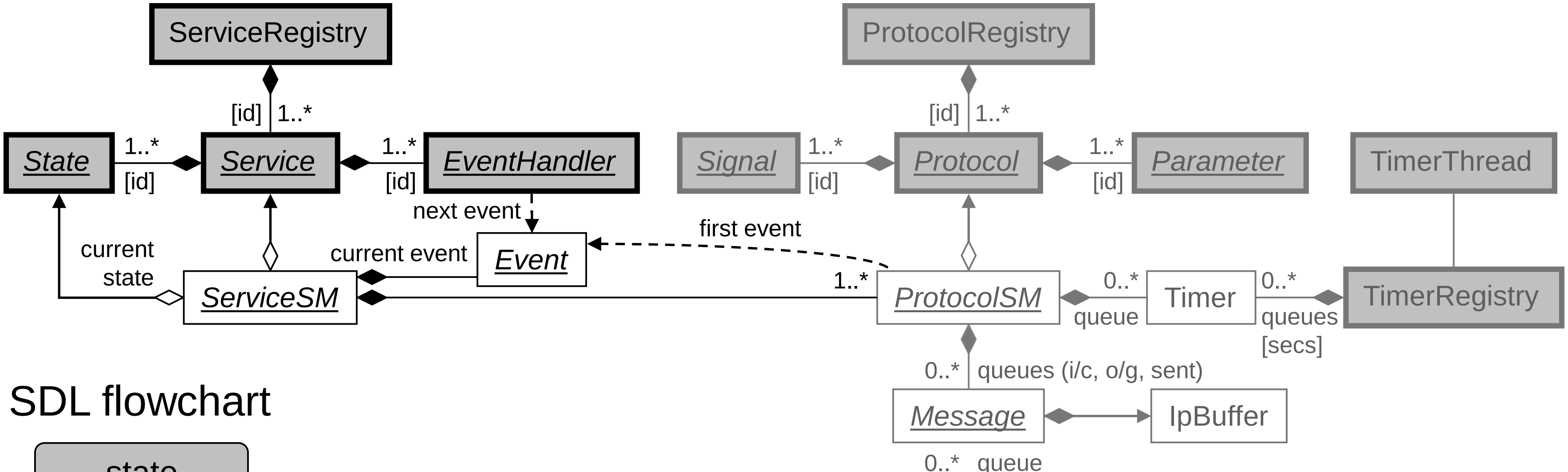
- each Timer is associated with a PSM
  - timeout message to PSM when timer expires
  - timer placed in registry (timewheel) serviced by timer thread
  - granularity = 1 second (soft real-time)



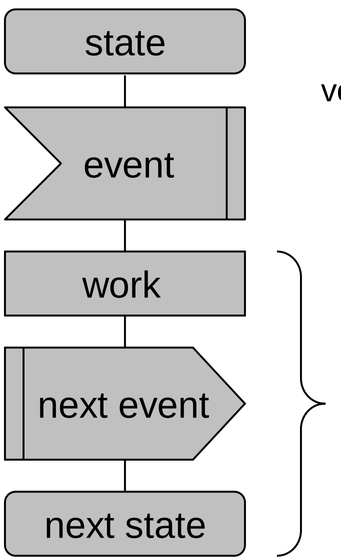
Service

- a Service defines a set of
  - Events: specify work that the service should perform
    - are objects (to allow arguments)
  - States: identify the event handler associated with an event
  - EventHandlers: react to an event in some state
    - perform work, including building outgoing message(s)
    - determine the next state and event
- a ServiceSM (SSM) is a per-user instance of a service
  - has a current state and event, and a next state and event
- invocation of event handlers done within framework
  - confines application software to event handlers and SSM functions (precludes service logic in state machine drivers)
  - simplifies patching of missing event handlers

Object Model (adding Services)



SDL flowchart




```
void ServiceSM::ProcessEvent(Event* currEvent, Event* &nextEvent):
    ...
    Event::Id eid = currEvent->Eid();
    EventHandler::Id ehid = currState_->Handler(eid);
    EventHandler* handler = service_->Handler(ehid);
    handler->ProcessEvent(*this, *currEvent, nextEvent);
    ...
```

(grey box)	singleton
(thick-bordered box)	instantiated during system start-up
italics	abstract class
<u>underlined</u>	subclassed by applications
◇	aggregation (doesn’t delete)
◆	composition (deletes)
——▶	unidirectional association
- - ▶	instantiation

Separation of Services and I/O

- use *Half-Sync/Half-Async* pattern to
  - eliminate I/O blocking in services
  - allow different services to use the same protocol
  - prioritize incoming work (for overload controls)

I/O threads → work queues → invoker threads

I/O threads

- one per IP port
- loop calls `recvfrom`, puts byte stream in an IpBuffer, passes buffer to InputHandler to add a message header that specifies the message's priority and target processing context

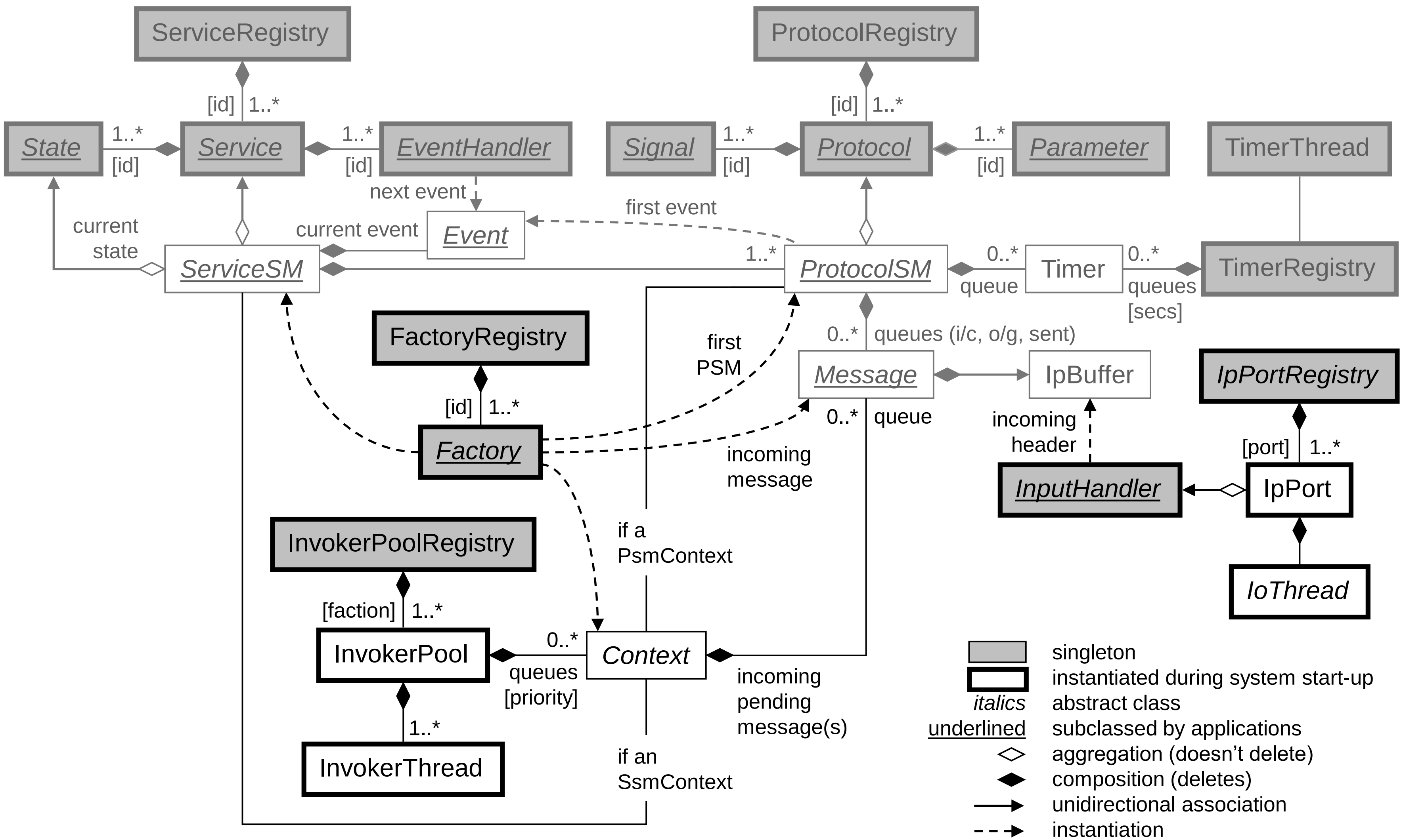
work queues

- *Queueing Layer*
- selected by scheduler faction (invoker pool) and message priority (a pool's work queue)
- contain contexts with pending messages

invoker threads

- one per scheduler faction (if no blocking)
- loop dequeues work in priority order and invokes context to process message
- runs each transaction to completion (locked)

Object Model (adding I/O)

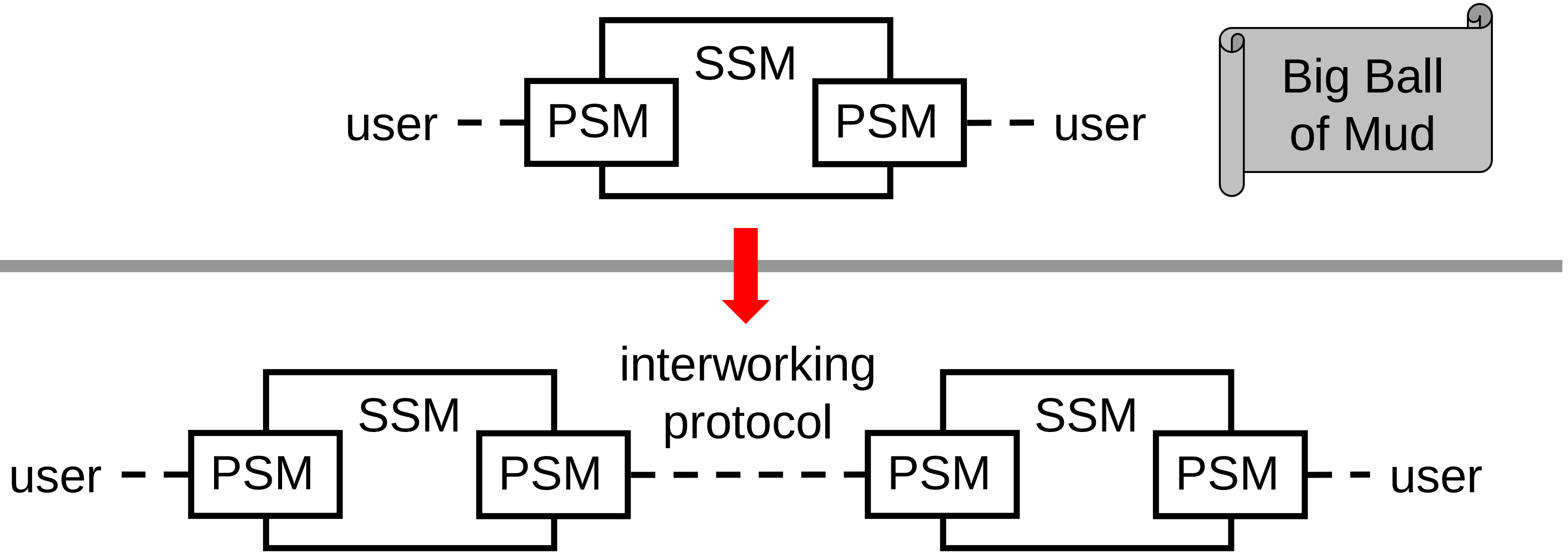


Separation of Services & Protocols

- many services use more than one protocol simultaneously
  - consequently, service states usually differ from protocol states
  - therefore have separate ServiceSM and ProtocolSM classes
- must map protocol input (signal) to service input (event)
  - implemented by PSM raising first event for SSM
    - usually, AnalyzeMessage event
      - SSM does **`switch(signal)`** to determine the actual event
    - rarely, ProtocolError event
      - when PSM detects a violation that requires SSM involvement
  - localizes code changes when enhancing or replacing a protocol
  - is a cleaner solution for *stimulus protocols*, in which the meaning of a signal depends on state
    - offhook signal maps to origination, answer, or reanswer event
    - mouse clicks and soft keys are also state-dependent

Separation of Sessions

- use separate sessions when connecting users
  - scalability: users can run on different processors
    - requires a database that maps users to processors
- sessions communicate using an interworking protocol
  - avoids O(n^2) development cost to interwork *n* user-network protocols



Message Preservation

- to reduce memory usage, default behavior is that
  - an incoming message is deleted at the end of the transaction in which it is processed
  - an outgoing message is deleted after it is sent
- however, the ability to save messages is useful
  - to reference parameters in an incoming message during a subsequent transaction
  - to retransmit an outgoing message if an acknowledgment does not arrive
- implemented by **`save`** and **`unsave`** functions
  - increment and decrement a counter
  - delete message when counter drops to zero

Type-Length-Value Message

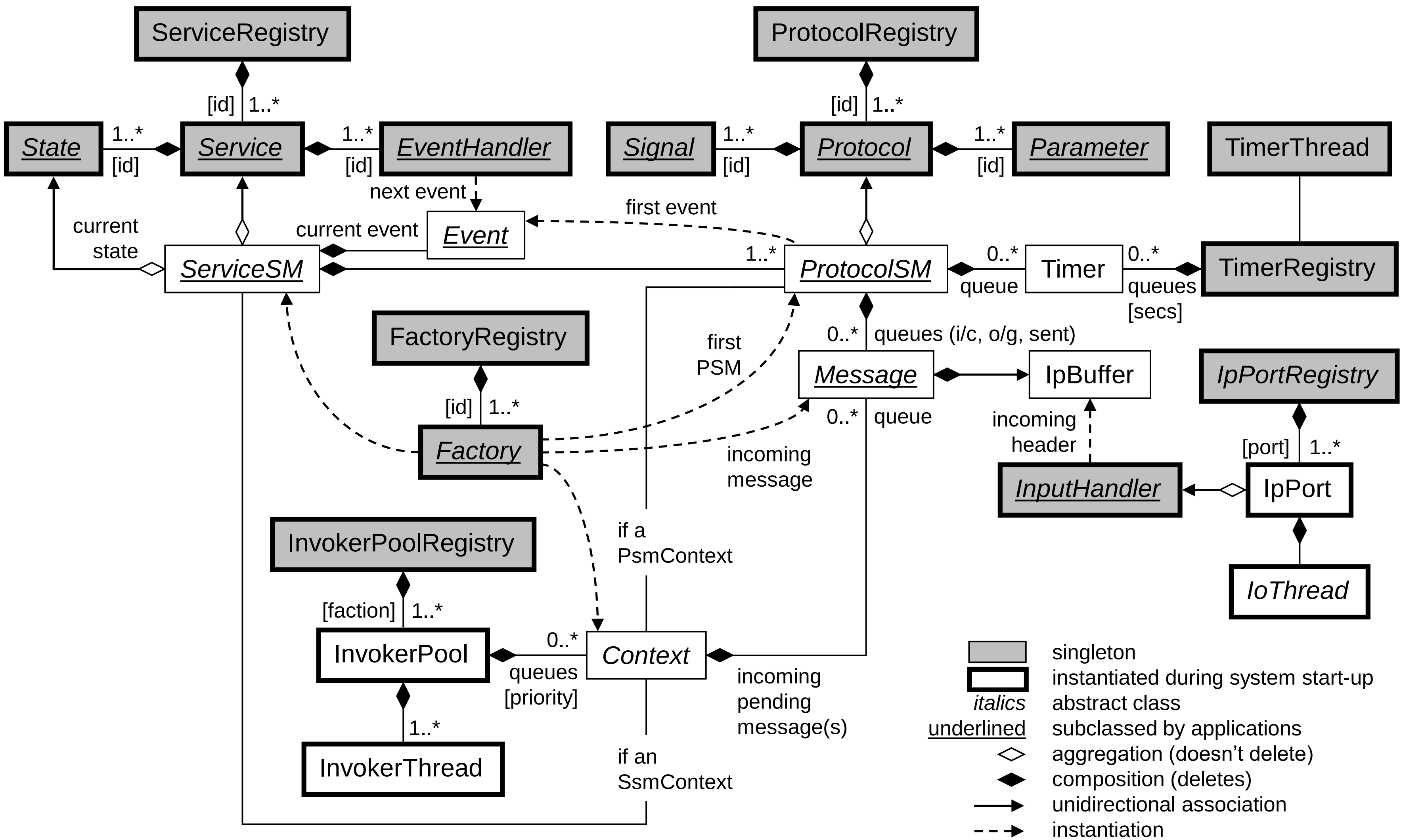
Technique	Synopsis
TLV Message	type (parameter identifier) + length + value (contents)—more efficient than text, ASN.1 or XML encodings
Parameter Typing	cast contents as a **`struct`** to improve reliability and readability
Parameter Fence	**`AddParm`** places pattern (e.g. **`0xaaaaaaaa`**) after parameter to detect trampling when next parameter is added
Parameter Template	**`AddParm`** uses parameter identifier to access a template that initializes the parameter
Parameter Dictionary	parse a message once, when it arrives, and construct a lookup table for fast access to its parameters; table is used by **`FindParm`**

- implement TlvMessage as a subclass of Message

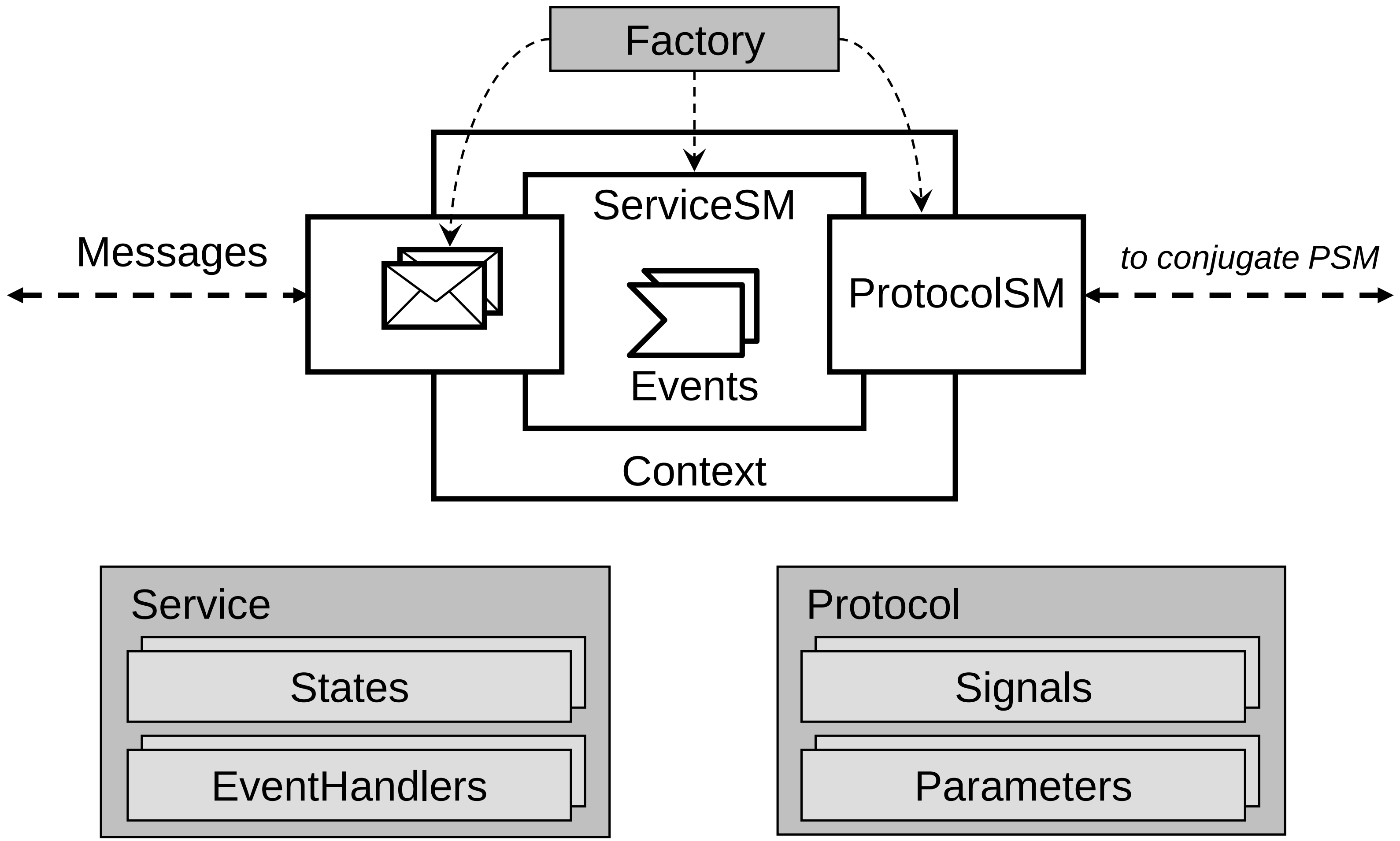
Session Processing Framework

- Object Model
- Run-Time View
- Class Responsibilities
- Application Subclasses
- Application Module Structure
- Changes in Terminology

Object Model



Run-Time View



Class Responsibilities

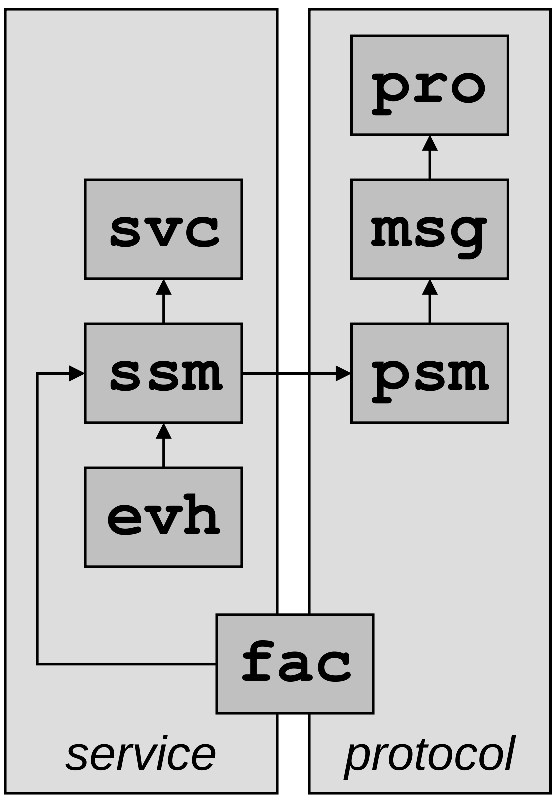
Class	Purpose
Service	provide registries for a service's states and event handlers
State	define the event handler to be invoked for each event that is legal in this state
Event	define an input to a service
EventHandler	handle one or more state-event pairs that can arise in a service
ServiceSM (SSM)	provide per-context data for a service that coordinates the behavior of PSMs
Protocol	provide registries for a protocol's signals and parameters
Signal	define a message type in a protocol
Parameter	provide additional information for one or more signals
IpBuffer	wrap a message (byte stream) and store source/destination IP addresses
InputHandler	add a header to an IpBuffer so that an incoming message can be routed
Message	manage an IpBuffer and provide functions to parse or build its contents
Timer	inject a timeout message to its PSM if it expires
ProtocolSM (PSM)	implement the state machine for the initiator or recipient role in a protocol
Factory	create incoming messages and the initial PSM, SSM, and context
Context	provide a scheduling *Façade* for a session's objects (messages, PSMs, SSM)

Application Subclasses

- Service, ServiceSM, and Factory: one per service
  - State: one per state
  - Event: one per event
  - EventHandler: one per state-event combination
- Protocol: one per protocol
  - Signal: one per signal
  - Parameter: one per parameter
  - Message: one per protocol, or even one per signal
  - ProtocolSM: one per protocol role (client and server, or initiator and recipient)

Application Module Structure

- services use protocols, but not vice versa
  - if S_1 and S_2 communicate using P, can then build S_1 and P into one load and S_2 and P into another



evh	EventHandlers
fac	Factory
msg	Messages
pro	Protocol, Signals, Parameters
psm	ProtocolSM
ssm	ServiceSM
svc	Service, States, Events
→	**`#includes`** relationship (transitive **`#includes`** not shown)

Changes in Terminology

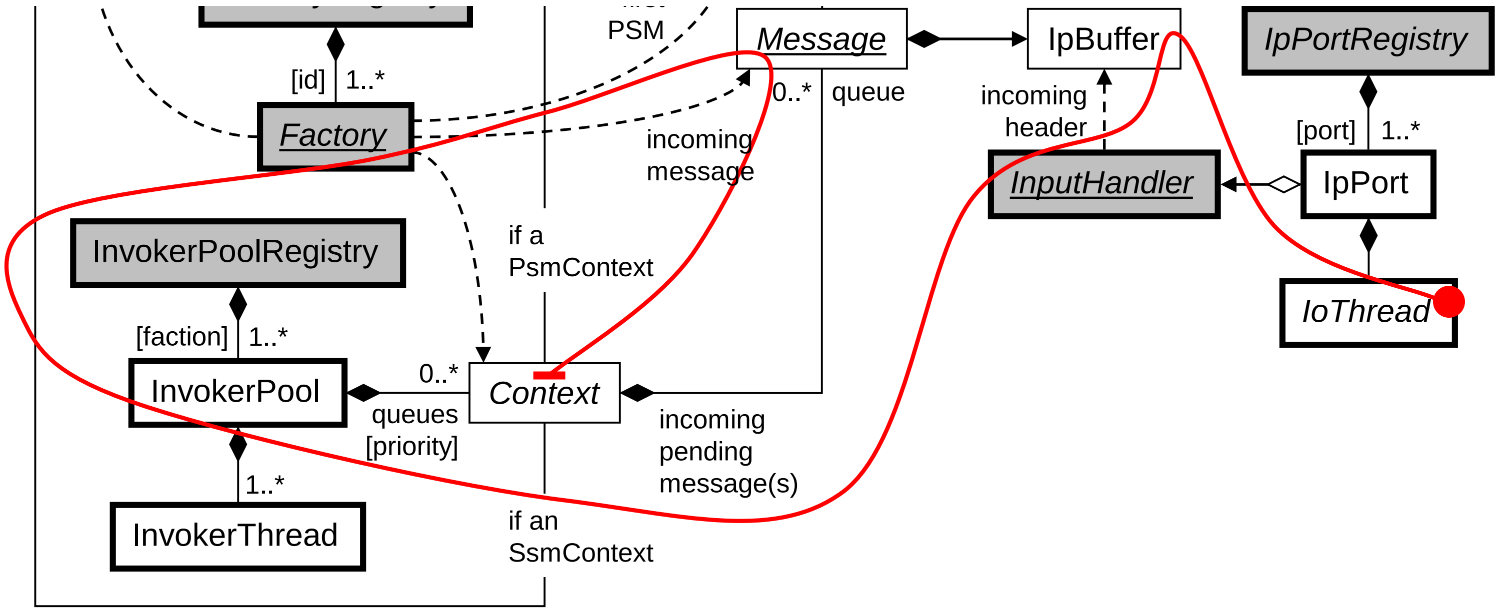
A Pattern Language of Call Processing	*Robust Services Core*
FSM	Service
PSM	ProtocolSM (PSM)
AFE	ServiceSM (SSM)
Agent	Factory
Transactor	Context
State, Event, EventHandler, Message	same
PFE, PFQ	not discussed (used to decouple supplementary services)
not discussed	other classes in object model

Session Processing Details

- Incoming Message Walkthrough
- Message Header
- Context Subclasses
- Context Determination
- Message Delivery Scenarios
- Message Crossing
- Transaction Walkthrough
- Database Design Guidelines
- Extreme Session Processing

Incoming Message Walkthrough

- IoThread receives a byte stream and places it in an IpBuffer
- IoThread passes IpBuffer to its InputHandler, which adds a message header for an external protocol
- IoThread passes IpBuffer to the InvokerPool associated with the IoThread's faction
- InvokerPool passes IpBuffer to the Factory specified in the message header
- Factory creates a Message to wrap IpBuffer
- in the case of an initial message, the Factory creates a Context, SSM, and PSM
- Factory queues the Message on the appropriate Context
- InvokerPool queues the Context on the work queue specified in the message header



Message Header

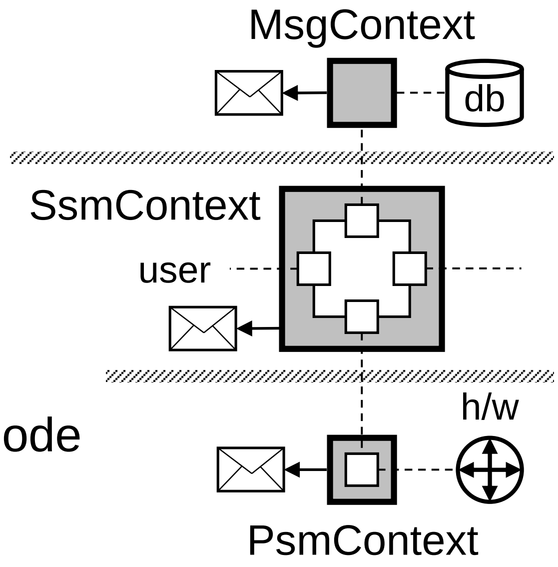
- interprocessor messages transported by UDP (IP address + port)
- sender supplies message header for all internal messages
- InputHandler supplies message header when external message arrives

```
struct MessageHeader
{
    LocalAddress    txaddr;         //  source address
    LocalAddress    rxaddr;         //  destination address
    unsigned int    priority : 2;   //  ingress, egress, progress, immediate
    unsigned int    initial : 1;    //  set if first message in protocol
    unsigned int    final : 1;      //  set if last message in protocol
    unsigned int    join : 1;       //  set to create PSM and join existing context
    unsigned int    protocol : 16;  //  protocol identifier (indexes ProtocolRegistry)
    unsigned int    signal : 16;    //  signal identifier (indexes Protocol)
    unsigned int    length : 16;    //  length of byte stream (payload)
};

struct LocalAddress
{
    unsigned int    pid : 32;   //  PSM identifier (indexes PSM object pool)
    unsigned char   seqno;      //  PSM incarnation number (see “Message Crossing”)
    unsigned char   fid;        //  factory identifier (indexes FactoryRegistry)
};
```

Context Subclasses

- a Context hides application objects from an invoker thread
  - incoming message queued on Context
  - Context placed on work queue
- three subclasses of Context
  - SsmContext has an SSM and PSMs
    - for a session
  - PsmContext has a single PSM
    - to access low-level functions in a remote node
    - example: controlling hardware
  - MsgContext has a single message
    - for a connectionless protocol (stateless request-response)
    - example: querying a database
- parallel Factory subclasses (SsmFactory, PsmFactory, MsgFactory) create corresponding Context subclass

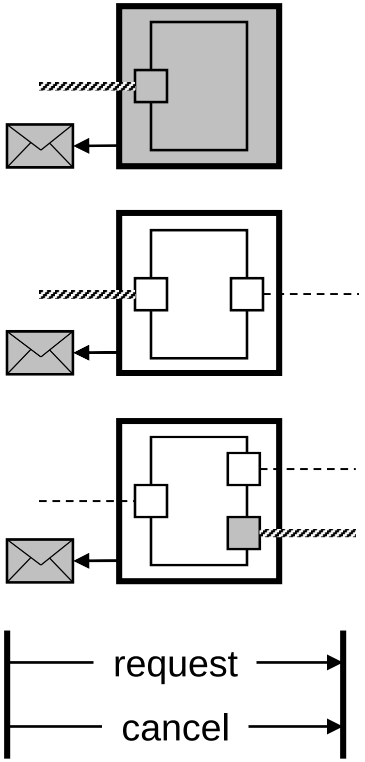


Context Determination

- incoming messages go through PSMs
  - but external protocols don't supply a destination PSM address!
- route external message to the correct PSM by using a protocol-specific parameter as a key
  - key could be source IP address and port, or user identifier
- external protocol must therefore implement a database that maps a key to a PSM
  - InputHandler looks up the PSM address in the database
  - if no PSM exists, InputHandler sets initial=true, and the Factory creates a new PSM and registers it against the appropriate key
  - PSM's destructor removes PSM from the database
- some external protocols echo a previously supplied value
  - avoid lookup overhead by getting the PSM address echoed

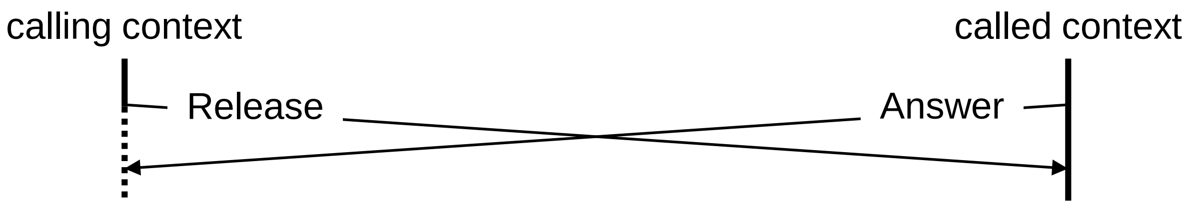
Message Delivery Scenarios

- message header determines how to deliver the message:
  - initial=true, join=false, PSM=nil
    - context does not exist
    - create context, and SSM and/or PSM if required
  - initial=false, join=false, PSM=non-nil
    - deliver message to existing PSM
  - initial=true, join=true, PSM=nil
    - SSM exists, but PSM does not
    - create the PSM that will receive the message
    - Factory uses protocol’s database to find the SSM
  - initial=false, join=false, PSM=nil
    - subsequent message *before acknowledgment*
    - source PSM has not yet learned the destination PSM’s address
    - destination node must find the destination PSM by using the source PSM’s address as a key (search all PSMs or implement a database)



Message Crossing

- must take this type of nasty situation into account:



- calling context is deleted after it sends Release message
- problem 1: PSM (object pool block) in calling context is reallocated before Answer message arrives
  - new PSM receives a spurious Answer message
  - fix by adding an incarnation number to PSM addresses
  - discard a message that contains the wrong incarnation number
- problem 2: billing records disagree on whether call was answered
  - add “call answered” flag to Release message for correlation

Transaction Walkthrough

- InvokerThread dequeues Context from work queue
- Context dequeues Message and queues it on destination PSM
- Context invokes destination PSM to handle Message
- PSM updates its state and raises AnalyzeMessage Event
- Context invokes SSM to handle Event raised by PSM
- SSM loops, invoking EventHandlers until next event is **`nullptr`**
  - first EventHandler is a message analyzer that maps the incoming signal to a service-specific event (see “Separation of Services and Protocols”)
- Context invokes each PSM that has a pending outgoing message
- PSMs update their states and send their messages
- Context deletes any PSM that is in the Null state
- if SSM is in the Null state, Context deletes the SSM and itself, else Context dequeues the next Message (if any) or returns to the InvokerThread

Database Design Guidelines

- databases used during session processing
  - reside in memory
    - improves throughput
  - support non-blocking queries
    - avoiding blocking operations improves throughput; also simplifies software by eliminating critical regions
    - queries support the option to return “record locked” rather than blocking until record becomes available
- user profile is cloned when session is initiated
  - can be fetched by request-reply message sequence
    - messaging is asynchronous, so blocking is OK in this case
  - clone eliminates subsequent blocking and allows the master to be modified in parallel with the session

Extreme Session Processing

Technique	Usage
object pools with audit	Contexts, PSMs, Messages, IpBuffers, Timers, SSMs, and Events each have their own pool
cooperative scheduling	I/O, invoker, and timer threads run to completion
asynchronous messaging	all Messages sent asynchronously
daemons	I/O, invoker, and timer threads created during start-up
proportional scheduling	I/O, invoker, and timer threads have factions
defensive coding	in applications generally; use Timer when waiting for ack
write-protected memory	most singletons (if data rarely changes after start-up)
safety net	I/O and invoker threads only lose current work item
obituaries	PSMs send/inject final messages (on death of context or node)
overload controls	work queues indexed by message priority
software error log	invoker threads can abort current work (context death)
object dump	display objects involved in current work (context death)
trace tools	tracing of messages and transactions