JAIN™ SLEE Tutorial

Serving the Developer Community

Swee Lim
Sun Microsystems

Phelim O'Doherty
Sun Microsystems

David Ferry
Open Cloud

David Page
Open Cloud

Java
Sun Microsystems

Sun microsystems

OpenCloud

Learn the JAIN SLEE application environment for building event oriented low latency and high throughput applications. Event handling and routing is an integral part of JAIN SLEE.

Container managed state variables and transactional semantics simplify concurrency control, consistency management, state replication to facilitate building of robust, scalable and highly available applications.

Presentation Outline

- Why create JAIN SLEE?
- Introduction to JAIN SLEE
- Implementation considerations

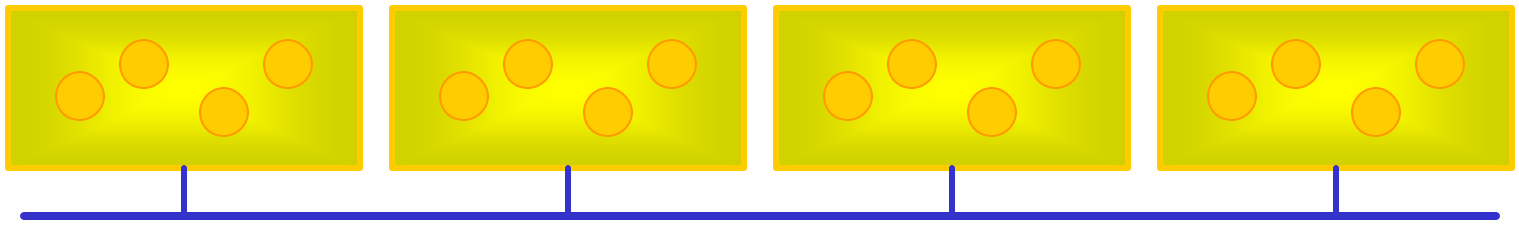
Why are Communications Applications Converging on Java Containers?

- Telco apps moving to component based architectures
- Desire to use Standard, Off-the-shelf container
  - Write-once, run-anywhere
- Container provides important infrastructure services
  - Higher level abstractions for State management, Transactions, Security, Resource pooling, ...
- Focus on core value-add application logic
- Leverage large community of Java developers
- Leverage enterprise development tools, test suites, ...

Time to market and reduced development cost

What is JAIN SLEE?

- SLEE = Service Logic Execution Environment
- Low latency and high throughput application server for event processing
  - Latency < 100 ms
  - 100's to 1000's of events per second
- Event oriented application environment
- Designed for stringent requirements of core network signaling application
- Designed for scalability and availability through clustering



Application Characteristics

	Communications	*Enterprise*
Invocations	Typically asynchronous - Events such as protocol triggers - Events occurrences mapped to method invocations	Typically synchronous - Database, EAI systems - RPC Calls
Event Granularity	Fine-grained events High Frequency	Course-grained events Low Frequency
Components	Light-weight fine-grained objects Short transient lifetimes - Rapid creation, deletion	Heavy weight data access objects Long persistent lifetimes
Data Sources	Multiple data sources - Location, context information - Provisioned data, cached from master copy	Database servers - Definitive master copy Back-end systems
Transactions	Light-weight transactions - For state replication demarcation - Faster completion and more frequent	Database transactions - Slower completion and less frequent
Computation	Compute-intensive - Processing is resource invocations & events	Database access intensive

Applications Characteristics

	Communications	*Enterprise*
Availability	3 to 5 9's	2 to 3 9's
Real-time	Soft real-time	
Deployment Distribution	Distributed deployment throughout network	Centralized deployment in small number of data centers
Nodes	1 to 4 CPUs	2 to 32 CPUs
Clusters	2 to 16 nodes	2 to 4 nodes

Applications characteristics drive Container Design!

JAIN SLEE Benefits

- High performing platform for event driven applications.
  - Supports simple and complex applications.
  - Applications deal with service logic only.
    - System issues handled by container i.e. threading, transactions
- Standard application framework.
  - Defined programming model
  - Object Orientated, asynchronous, robust and distributable
- Independent of underlying networks.
- Asynchronous support
  - Elaborate event distribution mechanism (with priority)
  - Maps events to method invocations on components
  - Creates component instances in response to initial events

SLEE reduces cost and improves time-to-market

JAIN SLEE Benefits

To Communication Developers

- Only industry standard Service Logic Execution Environment
  - Write-once, run anywhere for components
- Availability and scalability through clustering
  - Versus traditional primary-secondary
- Easy to develop robust components
  - SLEE replicates state and provides transactional semantics
  - Strongly typed component interfaces and profile data
- Point of integration for multiple protocols and resources
  - One container, multiple resources, protocols
  - Easy to integrate new new technologies

JAIN SLEE Differentiators

To EJB Developers

- Event oriented component model
  - Elaborate event distribution mechanism (with priority)
  - Maps events to method invocations on components
  - Event processing components with strongly typed interfaces
    - Event types received and sends
    - Private state and state shared with other components
    - Component instances have no external and permanent identity
    - Simple and dynamic event subscription model
  - Container manages component lifecycle and GC
    - Enables automatic instance creation and deletion
    - Model is aware of event producer and consumer relationships
    - Important for robustness (avoid dangling component instances)
- Profiles for provisioned data
  - Easy to define, provision, and access profiles

Options Evaluated during JAIN SLEE Design

	Pure SLEE container	SLEE mapped to EJB via code generation	SLEE re-specified as EJB + event processing library
Event processing support	Declarative (with optional Programmatic control)		Programmatic
SLEE Vendor provides	SLEE Container	EJB Container Event library Code Generator	EJB Container Event library
Component lifecycle management (GC)	Managed by SLEE Container	Managed by generated code, event library	Managed by application code, event library
Suitability for asynchronous applications	*High*	*High*	*Medium*
J2EE/EJB integration	*Medium*	*High*	*High*

- *Container designed for asynchronous applications should understand event models*
  - *Pure EJB containers lack native support for event models*
- *Required performance challenges current EJB containers*

SLEE & J2EE

- SLEE is a component model like EJB, Servlet or JSP, and is most similar to EJB.
- SLEE builds upon concepts in J2EE technologies but is a specialized component model for event driven applications.
- The SLEE can be implemented independent of J2EE and used stand-alone without requiring a J2EE adjunct
  - not dependent on J2EE technologies to outsource critical functions like concurrency control or failure resilience
- SLEE is not a component of J2EE and is not the equivalent of J2EE

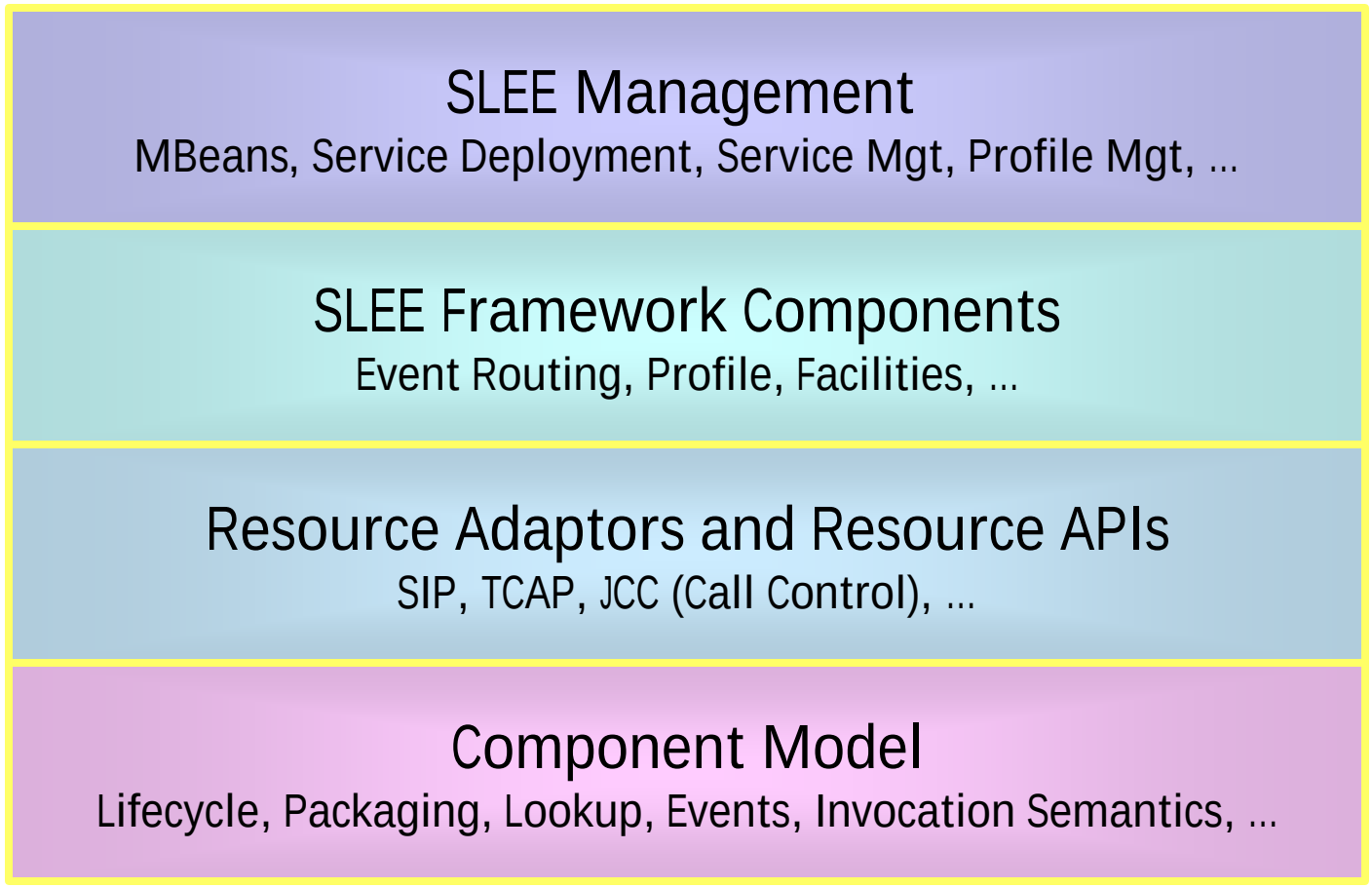
SLEE, EJB & JMS

- SLEE has an built-in event model that is part of the component model
  - JMS is external to the EJB component model, it is more like a service
- SLEE allows dynamic creation of activities (like topics and queues) and dynamic association of event sources and event sinks to these activities
  - Binding between JMS queues, topics into the EJB runtime is static, there is no way to dynamically create new queues, topics and dynamically update the event routing mechanism under program control
- SLEE event model facilitates component garbage collection
- JMS can be integrated into SLEE through a Resource Adaptor just as in EJB via a Connector

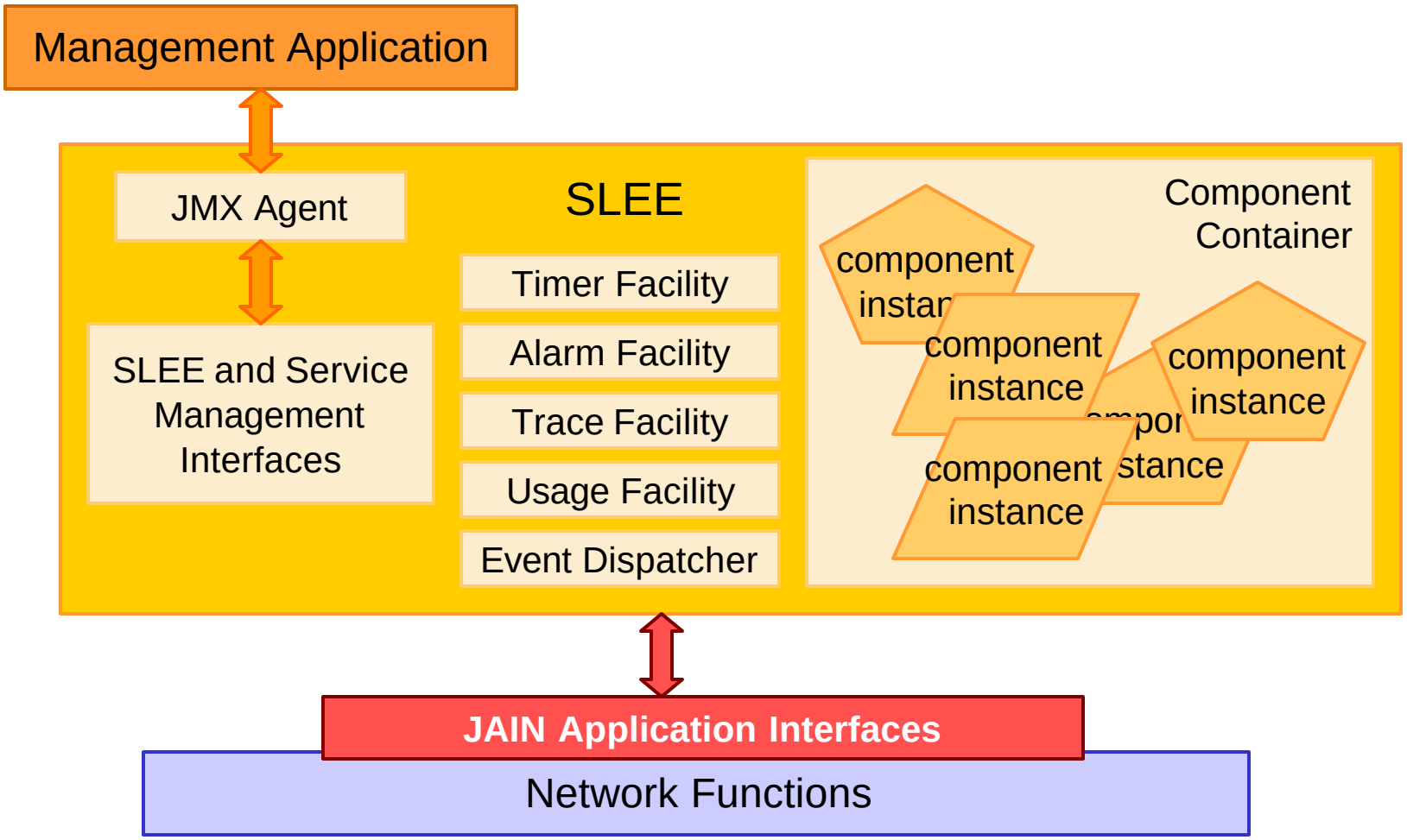
Presentation Outline

- Why create JAIN SLEE?
- Introduction to JAIN SLEE
- Implementation considerations

Major Subsystems



SLEE Architecture



Types of Components

- Event Types
  - Declares an event type
  - Determines how the event is routed and event class of the event
  - Every event has an event type
- Service Building Block (SBB)
  - Contains application and service logic (like an EJB)
    - Event handler, local interface, life cycle, callback method implementations
  - An SBB can be composed from one or more “child” SBBs
- Service
  - Specifies information needed to create SBBs to process “initial events”
- Profile Specification
  - Specifies the schema of Profile Tables, Profile Table contains Profiles
  - Verification logic for provisioned data in a Profile

SLEE Event Model

In a managed environment like SLEE direct references are only acceptable for use within a single method body. The developer should be concerned with attaching to an event bus, and detaching from an event bus within a transaction.

Event Model

- SLEE uses the publish/subscribe model for event distribution.
- SBB's attach and detach to activity contexts for event distribution.
- The SLEE architecture does not allow SBB's to register themselves explicity as listeners of a resource.
  - Does not follow JavaBean event model
- In the SLEE model the resource adaptor has outsourced event subscription and arbitration to the SLEE
  - allows resource adaptors to be consistent in their event model

Event Consumers & Producers

The SLEE understands the relationship between event producers and event consumers

- SLEE can properly manage the lifecycle of SBB components.
- SLEE can be effective in managing and optimizing event subscription and filter setup and event distribution
- SBB's can not pass remote-references to resources, as the SLEE wouldn't know whether or not the SBB is referenced and could not destroy the SBB.
- The SLEE may maintain a GC thread or process that looks for attachment count of zero for all SBB entities in order to garbage collect them.

Event Routing Behavior

- Events are asynchronous
  - Therefore firing an event is not a blocking call.
- When an Event is fired within a transaction, it is placed into an Event queue and the fire method returns.
  - For example if a SBB fires an event that it should receive itself, the event is queued for delivery and the fire method returns immediately.
  - The SBB will get the event based on the routing policy.
- For each event type received by the SBB, the SBB developer must implement an event handler method.
  - Deliver event by invoking appropriate event handler method of the SBB
- If multiple SBB entities interested in the same event, the SLEE defines event delivery order
- For each type of event fired by the SBB, the SBB developer must declare an abstract fire event method
  - This abstract fire method is implemented by the SLEE at deployment time.

Event Subscription

- Declarative event subscription
  - SBB declares event types received
  - Root SBB declares initial event types
- The SLEE enables event filtering mechanisms to reduce the amount events received by the SLEE.
  - Event filters may exist within the SLEE or can be pushed to the resource
  - The SLEE and resource adaptors work together to update event subscription on resources automatically
  - An SBB entity can mask and unmask by event types for each attached Activity Context
    - Declare which event should be masked
    - SBB event mask in the SBB deployment descriptor information.

Event Filtering

- The SLEE may create its own event filter objects
  - The resource cannot know ahead of time the events the SLEE is interested in.
  - For example, using JCC the call processing platform passes all events to the SLEE.
- The SLEE may instruct the resource of event filtering information via its provider object
  - For example, JCC would deploy the appropriate JccListeners into the switch, providing address and event information through the event filter mechanism of JCC.
  - The JCC provider implementation is able to modify its trigger database when the SLEE instructs the provider to create the event filter objects from the information contained in the event filters.
- The SLEE specification does not mandate that the resource platform and the SLEE share a common database schema.
  - The trigger view inside the resource platform and the subscriber mechanism inside the SLEE don't have to be strictly in sync because of the timing of calls entering the system.
  - It is also possible for a SLEE to be tightly integrated with a particular resource platform and use a tightly coupled filter mechanism.

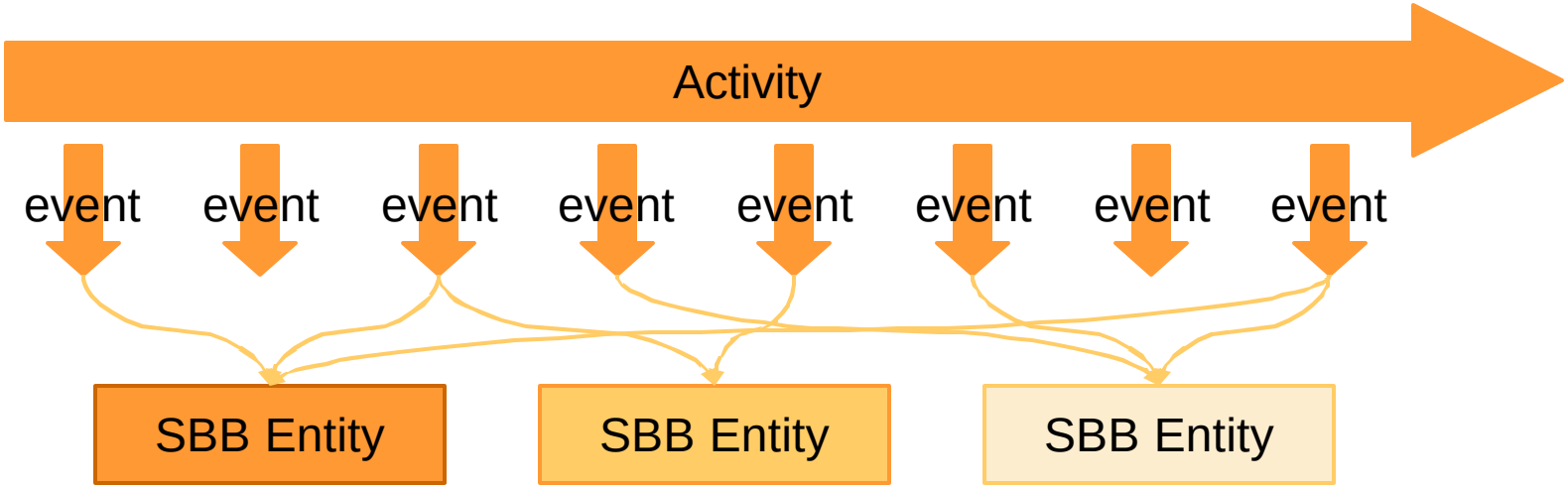
Custom and Non-Custom Events

- Custom and non-custom events should not be treated differently by the event router.
- Custom events are defined by SBB developers
  - Custom event requirements are defined by the SLEE specification.
  - Custom events are used for inter-component communication
- Non-custom events are defined by the SLEE and resource adaptors
  - their requirements may be relaxed or more restrictive depending on the SLEE implementation and the underlying source of the events.
  - For SLEE vendor flexibility in adapting existing resources to the SLEE, non custom events do not have to follow the extra rules defined by the SLEE for custom events.

Activity and Activity Context

Activity

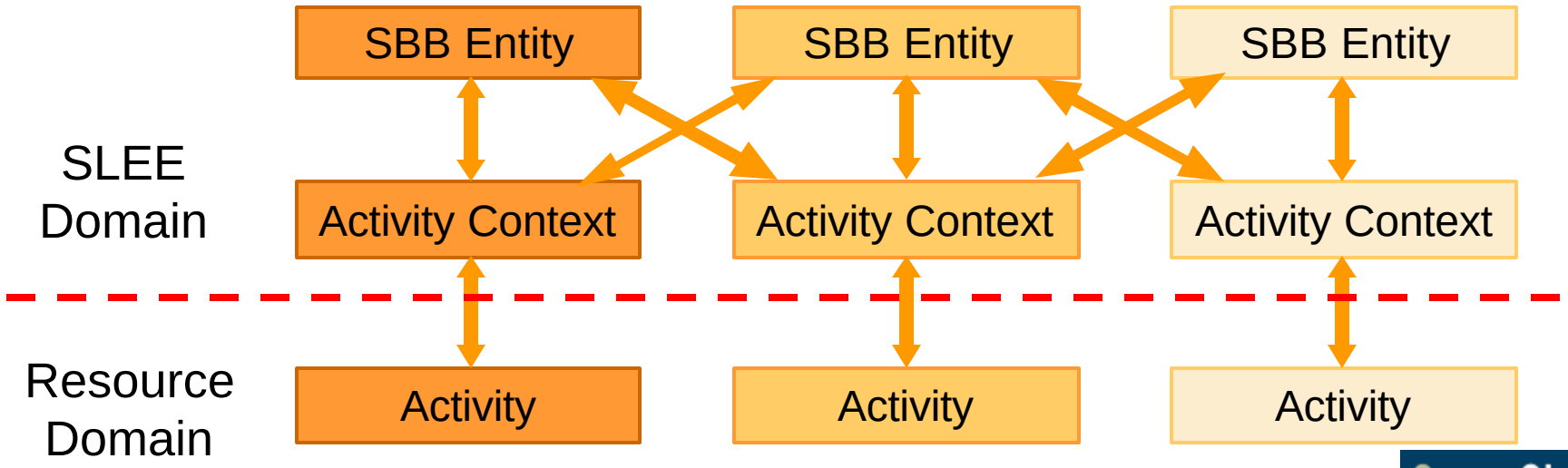
- Abstraction for a related stream of events
- Examples
  - Call object emitting call connected, disconnected, ... events
  - Mobile location report object emitting location update, ... events



Events are routed to SBB entity that are interested in them.

Activity Context

- SLEE domain object to encapsulate Activity defined by resources
- An event channel on which events are fired and delivered on
  - Like a JMS topic
- Holds shared state regarding the Activity
- Many-to-many attachment relationships
- SBBs only receive events from attached Activities



Activity & Activity Context

- An Activity object is a resource object.
  - Represents some resource system that encapsulates a flow of events.
- A single Activity can only process one event at a time.
- An ActivityContext is the SLEE representation of a resources Activity.
- SBBs receive events on an ActivityContext.
- An ActivityContextInterface is a SBBs view of the ActivityContext.
  - SBBs may read and write state in an ActivityContext.
- Each ActivityContext has a 1-2-1 relationship with an Activity object.
- Each SBB defines a Java interface that represents their view onto the state stored in the ActivityContext.
  - This provides type safety and reduces bugs as a contract is defined between components, activity contexts and the state being modified.
- The appropriate ActivityContext is gained by passing an Activity to the ActivityContextInterfaceFactory.

Activity Context and Events

- Activity Context is a means to relate event producers and event consumers.
  - Activity Contexts behave like an event bus. Events are fired and received on Activity Contexts.
  - SBB entities attached to the event bus receive the event if it does not explicitly mask the event.
- A consumer must be attached to an Activity Context to receive events fired into the Activity Context.
  - attaching to an Activity Context serves the same function as adding a listener.
- SBB entities listening to the same type of event will typically be attached to different Activity Contexts
  - will only receive events fired on the Activity Contexts to which they are attached.

Attach/Detach on Activity Context

- Decouples event consumer and producer so container is in control.
- Allows SLEE to prioritize event delivery.
- Enables transacted operations.
- Replaces object references with some 'distributable identity' object.
- Replaces the add/remove Listener concept in the JavaBean event model for distributed systems
  - JavaBeans is a popular development pattern but it is not sufficient for a robust distributed system.
    - This implies the event source should maintain references (or pointers) to the event destinations
    - Turning local object invocations into remote invocations to add distribution to a system has performance impacts.
- The industry is moving to publish/subscribe messaging based integration.
  - The listener approach is the equivalent of point-to-point enterprise integration.

JCC Activity Context Example

- Activity objects defined by a JCC resource are JccConnection and JccCall.
- Assume a new call comes into the SLEE. This call has one call leg that an SBB is interested.
  - The JccConnection Activity object passes a CONNECTION_ALERTING event to the ActivityContext of the activity representing the call leg and the SBB receives the event.
- The SBB decides it wants to disconnect the connection. To do so the SBB needs the Activity object from the ActivityContext:

```
public void onAlertingEvent(JccConnectionEvent event, ActivityContextInterface ac){
     JccConnection connection = (JccConnection)ac.getActivity();
     connection.release();
}
```

- Assume that a new call is made from an SBB:

```
JccProvider provider = (JccProvider) new InitialContext.lookup("location");
JccCall call = provider.createCall(args);
```

- To receive events on this call, the SBB must get an ActivityContext of the new Activity, via the ActivityContextInterfaceFactory of the resource:

```
ActivityContextInteface ac =
          JccActivityContextInterfaceFactory.getActivityContextInterface(call);
ac.attach(sbbLocalObject);
```

Service Building Block

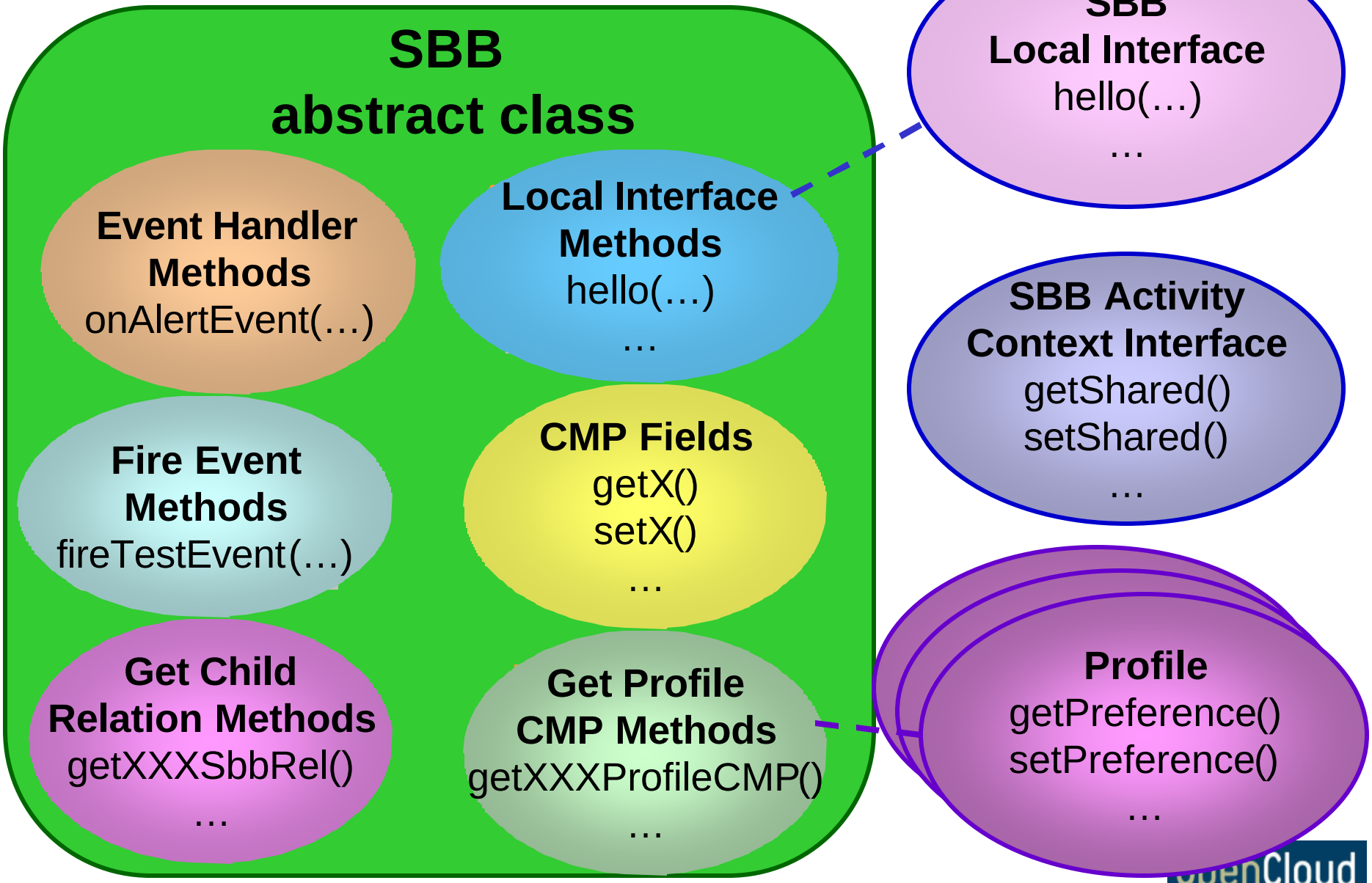
SBBs and Services

- An SBB is a service building block
  - It is a programmatic component of a service
- A Service instance can contain a single SBB or many SBB instances of different SBB types.
- The same SBB type can be included in multiple Service types.
- A single SBB can only process one event at a time.
- Multiple SBB's belonging to the same Service can process events in parallel.

Developing an SBB

- The SLEE specification defines the Sbb Interface
  - The SBB interface is the base SBB interface defined by the SLEE specification and is included in the SLEE jar file
- SBB developer implements SBB abstract class by implementing SBB interface
  - Each SBB created by the SBB developer, must provide a SBB abstract class that implements the base SBB interface.
  - The SBB abstract class contains the developer provided code for the life cycle callbacks, event handler methods, and abstract fire event methods.
- The SLEE deployment tool generates the SBB concrete class by implementing the SBB abstract class.
  - The SBB concrete class is generated by the SLEE when the SBB is deployed into the container.
  - The SBB developer doesn't implement the SBB concrete class.
  - The deployment tool implements the SBB concrete class by extending the SBB developer provided SBB abstract class.

Service Building Block (SBB)



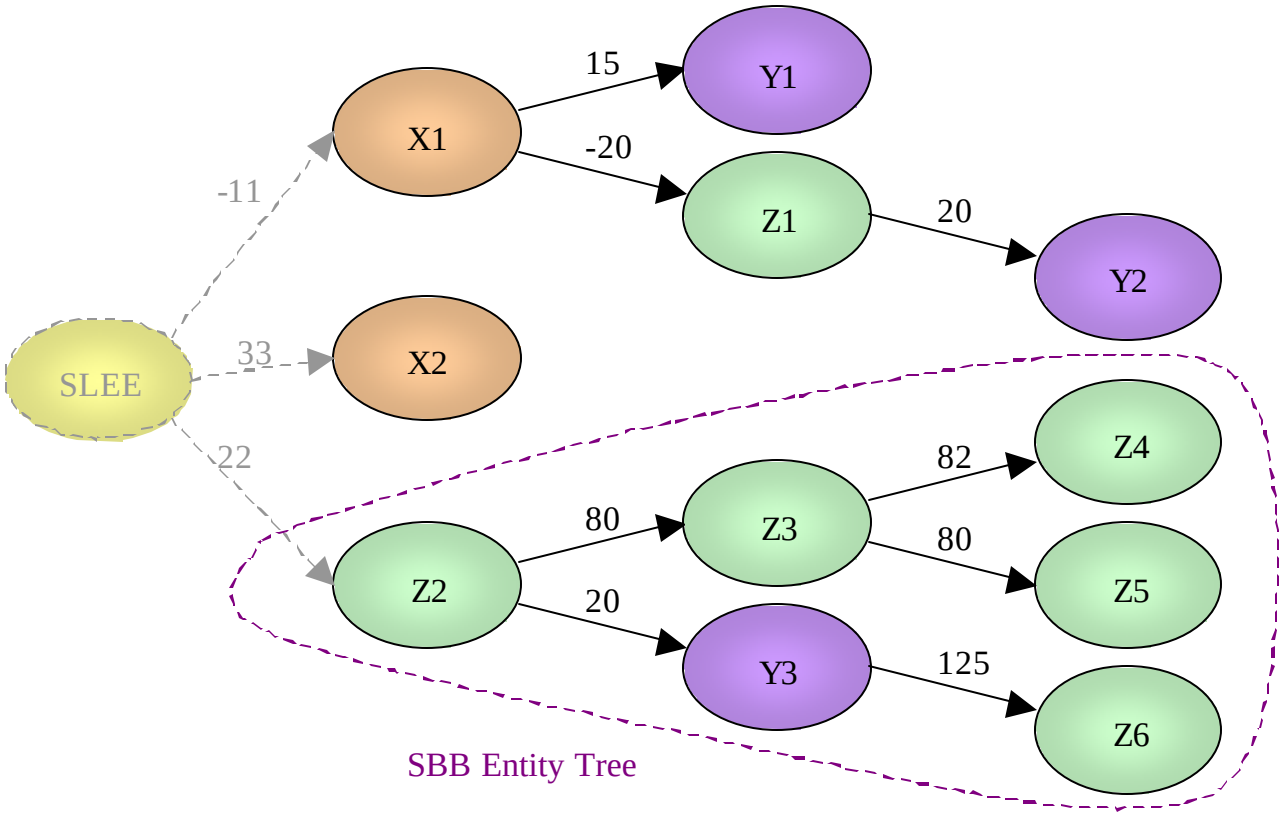
SBB Entities

- An SBB entity is an instance of an SBB component.
  - It is a logical entity.
  - In the simplest form there is one SBB entity for a service instance.
- An SBB represents the persistent per-instance state of an instance of the SBB component.
  - there exists some managed representation of the state that has transactional properties.
  - the managed representation could be in process memory, replicated memory, on disk storage, tape backup, etc.
  - The per-instance state of an SBB instance is defined by the CMP fields in the SBB abstract class of the SBB component.
- A root SBB entity is the root node in an SBB entity tree and is an instance of a root SBB.
  - It is instantiated by the SLEE to process its initial event.

SBB Entity Trees

- An SBB entity always exists within an SBB entity tree.
- An SBB entity tree is always within the context of a service 'instance'.
- The service is a deployable unit in the SLEE
  - This deployable unit is how the SLEE knows to instantiate components at runtime to handle 'phone calls' or activities.
- The root SBB of the service handles the initial event
  - Dependent on the initial event selector in the SBB deployment descriptor

SBB Entity Trees



A *SBB entity tree* shows parent child relations among instantiated *SBB entities*

SBB entity

parent to child relation

-20 actual priority

SLEE is logical parent of all *root SBB entities*

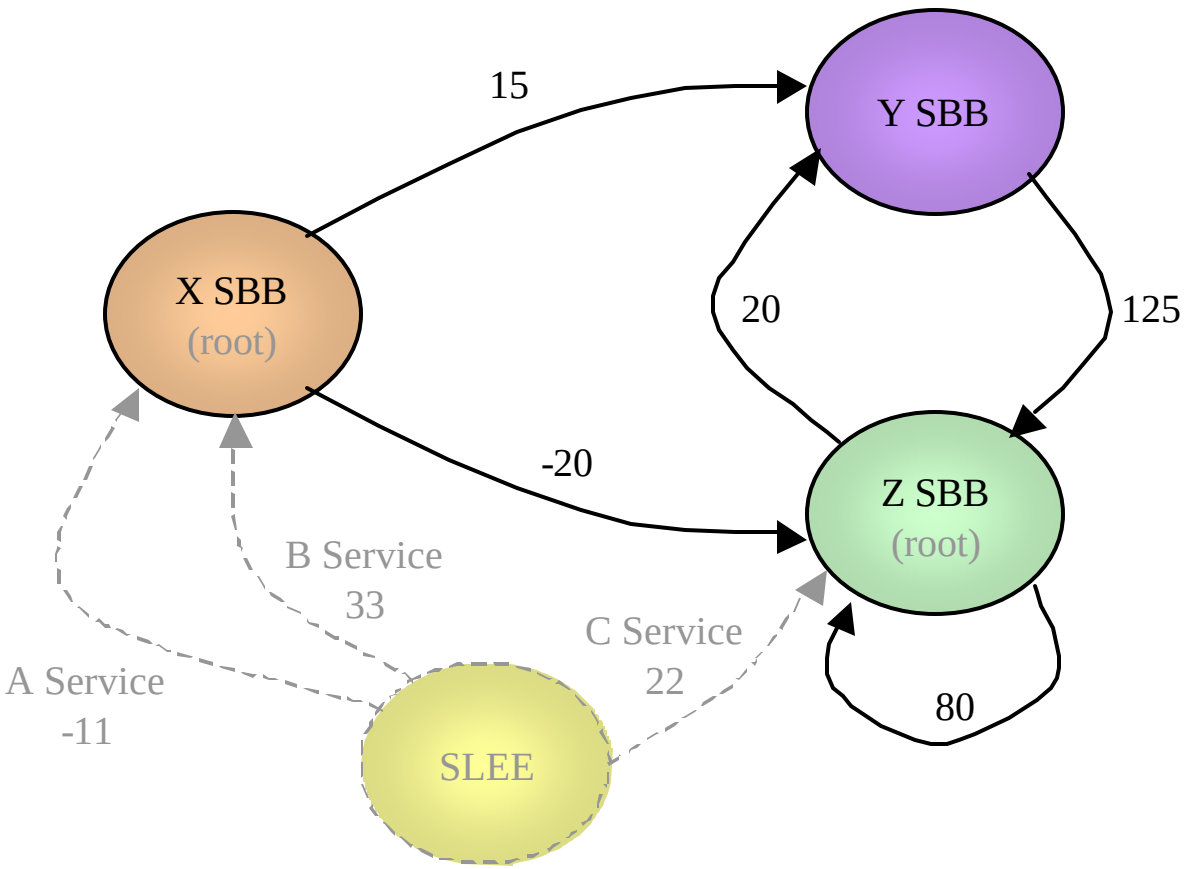
Root SBB entities are instances of SLEE instantiated *root SBBs*

- Event delivery order (by priority, parent then child)
- Cascading removal of SBB entity sub-tree

Parent & Child SBB Entities

- The SBB entity model is cyclic
  - The SLEE creates the parent SBB entity.
  - The parent SBB creates its children SBB's entities and so on.
- The parent SBB has access to the child relation object
  - Use the "get child relation object" method declared by the parent SBB developer in the parent SBB abstract class.
- The parent SBB entity can get an SBB local object that represents itself
  - Use the getSBBLocalObject method in its SbbContext object.
- The parent SBB entity can pass this SBB local object to any child object that it creates
  - Typically, this is done by defining a method in the child SBB's local interface intended to be used by the parent SBB for passing information to the child SBB entity.

SBB Composition



SBB can be composed from one or more *child SBBs*

A *SBB graph* shows parent child relations among *SBBs*

SBB

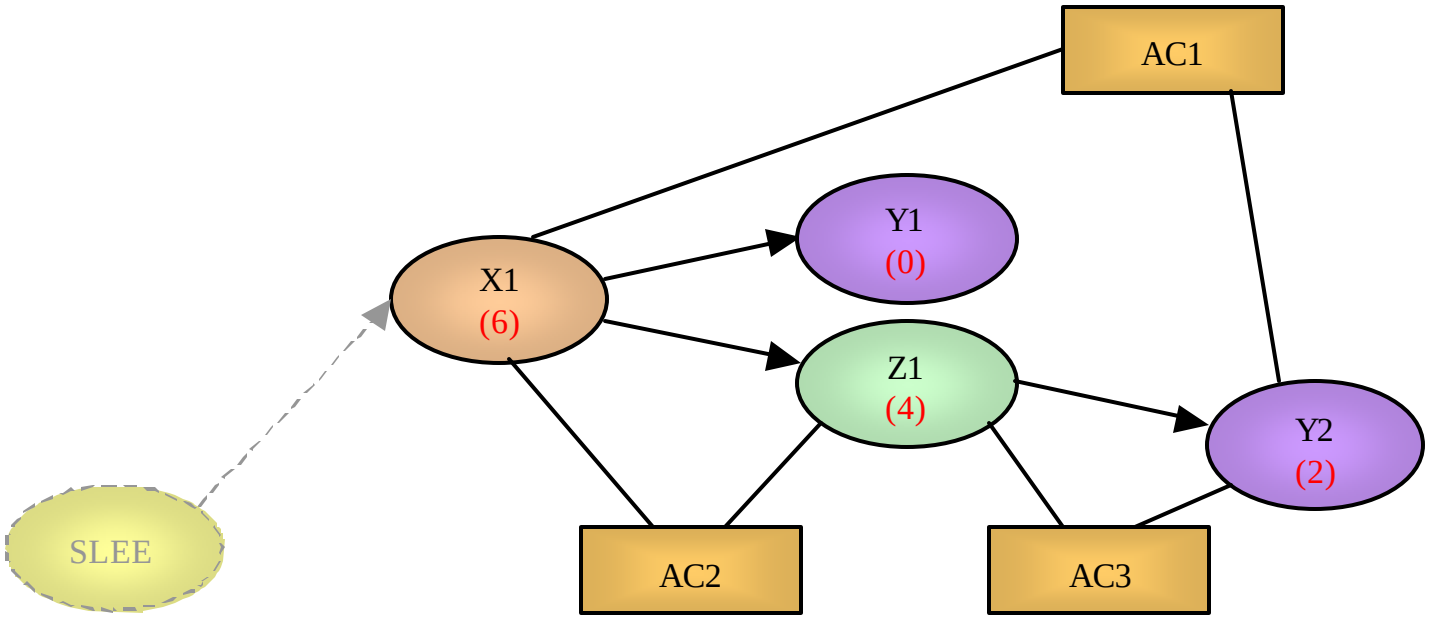
parent to child relation

-20 default priority

SLEE is logical parent of all *root SBBs*

Root SBBs can be instantiated by the SLEE.

Attachment Count & Entity GC



- Attachment count is recursive
- Attachment count facilitates automated SBB entity garbage collection

(In addition to native Java Object garbage collection)

Creating Child SBB Example

- A CallForwarding application may well be composed of a Call application and a Forwarding application.
  - This may be implemented as a parent SBB (CallForwardingSbb) with two child SBB's, i.e. CallSbb and ForwardingSbb respectively.
- In the parent SBB (CallForwardingSbb) you must define two methods which will enable the retrieve the ChildRelation objects:
  - ```
    public abstract class CallForwardingSbb implements Sbb {
          public abstract ChildRelation getCallSbb();
          public abstract ChildRelation getForwardingSbb();
    }
    ```
  - The SLEE implements the child-relation accessor methods, hence the methods are abstract in the SBB class.
- The SBB application developer invokes the create method on the ChildRelation object:
  - ```
    public class CallChildRelation implements ChildRelation {
          public SbbLocalObject create() ..... {
                SbbLocalObject local = new CallSbbLocalObject();
                return local;
          }
    }
    ```
  - The SLEE will create a new SBB entity, the sbbCreate lifecycle method is called on an SBB object, as part of the creation of the SBB entity. The `SbbLocalObject` of the SBB entity is returned to the application.

SBB Objects

- An SBB object is an instance of an SBB abstract class.
  - It has a life cycle i.e. does not exist, pooled and ready.
- An SBB object is 'bound' to a particular SBB entity for the duration of a method invocation.
  - When an SBB object is in the Ready state, the SBB object has been assigned to an SBB entity.
  - It executes on behalf of the SBB entity and it can access the persistent state of the SBB entity.
  - After that invocation it is possible that the SBB entity still exists, but there is no SBB object 'bound' to it.
- A collection of SBB objects might represent an entity.
  - These objects could be in the same or different JVM, but they can act "on" the same SBB entity.

SBB Local Objects

- The SLEE architecture permits application control, dependant on the component state machines and via component use of SBB local Interface
  - Enables application control over which components are attached and detached from the Activity Context, hence which components receive events.
  - Replacement for the dynamics of add/removeListener pattern in a highly available, concurrent, distributed environment.
  - Application has enough knowledge to do dynamic arbitration because the components are co-written or at least have enough of an API to allow the control.
- A SBB local interface is basically a local version of a remote proxy, not a reference to the SBB object
  - Operations on a SbbLocalObject are essentially passed through to the SBB object that is representing the SBB entity.
  - Using the control delegate and control coordinator pattern, a component receives an event, decides that it should tell its coordinator that it has finished with the event via the local method invocation
  - The control coordinator is invoked in the same transaction and attaches another SBB and calls `startWithActivity` on that SBB.
  - The transaction commits and the exchange of attach and detach is atomic.

SBB Entity & SBB Objects

- Depending on sizing requirements of a particular service, there might be n SBB objects and m SBB entities at a particular point in time.
  - If n is less than m, there will be m - n SBB entities that have no SBB objects 'bound' to them.
- The SLEE must understand the event consumer/producer relationship at the entity level rather than object level
  - due of indirection between SBB object and SBB entity, i.e. SBB entity is distinct from SBB object.
  - The state of the SBB entity is not directly related by the state of the java object that is used for a particular invocation of that entity.
- For example a SBB entity may be stored using a database, the SBB object can be garbage collected but the entry still exists in the database.
  - The SLEE must ensure that the database state is not 'leaked' because a programmer loses reference to it.
  - Referencing mechanisms in Java like soft references can't be used for SBB entity garbage collection

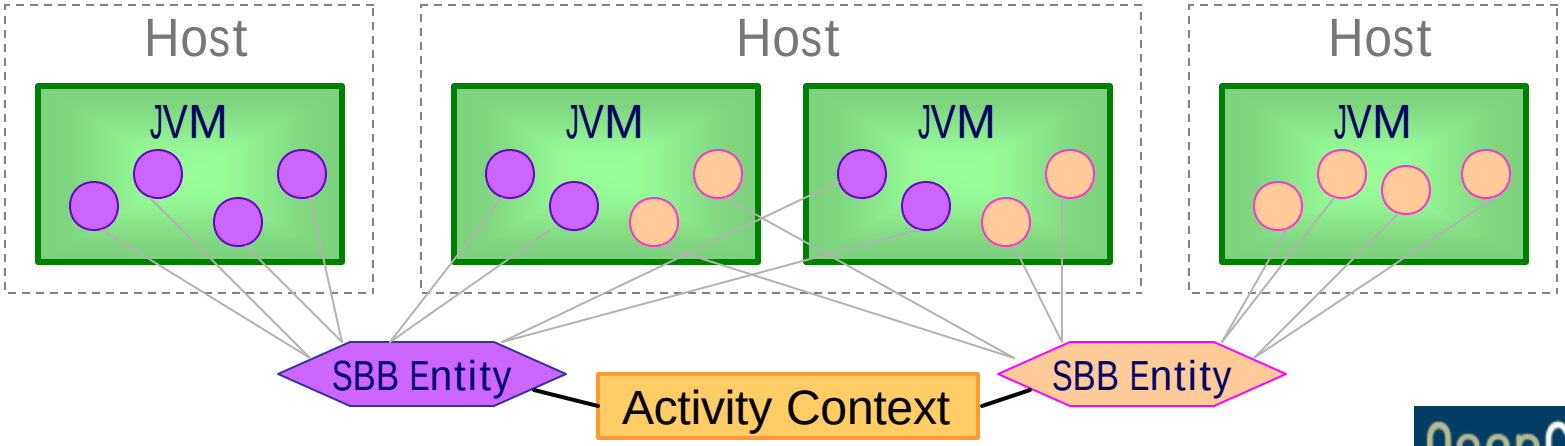
SBB Entities & SBB Objects

SBB Entity

- Instance of *SBB*
- A logical entity
- Represents persistent state of SBB (as defined by CMP fields)
- Maintains relations with other entities, e.g. attached Activity Contexts, parent and child SBB entities, its Service entity

SBB Object

- An *SBB abstract class* instance
- A Java Object
- Caches persistent state of an *SBB entity*
- Has a life cycle (may be pooled)
- May cache different SBB entities through its life time



SBB Priority

- SBB priority is defined to allow simple arbitration between independent components
  - That is components that are not co-written by a developer
  - The SLEE does not require that all interactions are codified.
- For example, two components that are not written together may happen to receive events on the same activity. There are two options:
  - Generally the default priority mechanism is used to deal with this interaction.
    - The component with the highest priority should run first.
  - The operator/administrator may if necessary codify the interactions

SBB Attribute Aliasing

- Each SBB specifies the attributes that it is willing to share with other SBB entities in the SBB Activity Context Interface.
  - By default, these attributes can be shared among SBB entities of the same SBB component.
  - They are not accessible by SBB entities of other SBBs to avoid unintentional sharing resulting from two SBBs developed independently selecting the same attribute name for two semantically distinct attributes.
- Activity Context attribute aliasing directs the SLEE to make the attributes declared in different SBB entities behave logically as a single attribute.
  - This logical attribute can be updated through any of the aliased attributes' set accessor methods.
  - Changes to the logical attribute are observable through any of the aliased attributes' get accessor methods.
- Shareable attributes defined in an SBB Activity Context Interface of an SBB are not "shared" with other SBBs entities unless explicitly aliased.
- For example,

```
SBB1 defines an attribute 'forwardCounter'
SBB2 defines an attribute 'followCounter',
Semantically they mean the same.
They can both be aliased to 'counter'
And shared in the SBB's Activity Context.
```

Counter SBB Example

- An SBB abstract classes for the counting SBB.
  - The counting SBB uses CMP state to store the current value of the count.
  - This CMP field is an integer.
  - The counting SBB has four methods on its SbbLocalObject interface called `void increment(), void decrement(), getValue()` and `void reset()`.
- The implementation of these methods in the Sbb abstract class are:

```
public void increment(){setCounter(getCounter()+1);}
public void decrement(){setCounter(getCounter()-1);}
public int getValue(){return getCounter();}
public void reset(){setCounter(0);}
// accessor methods for a field called counter
public abstract void setCounter(int val);
public abstract int getCounter();
```

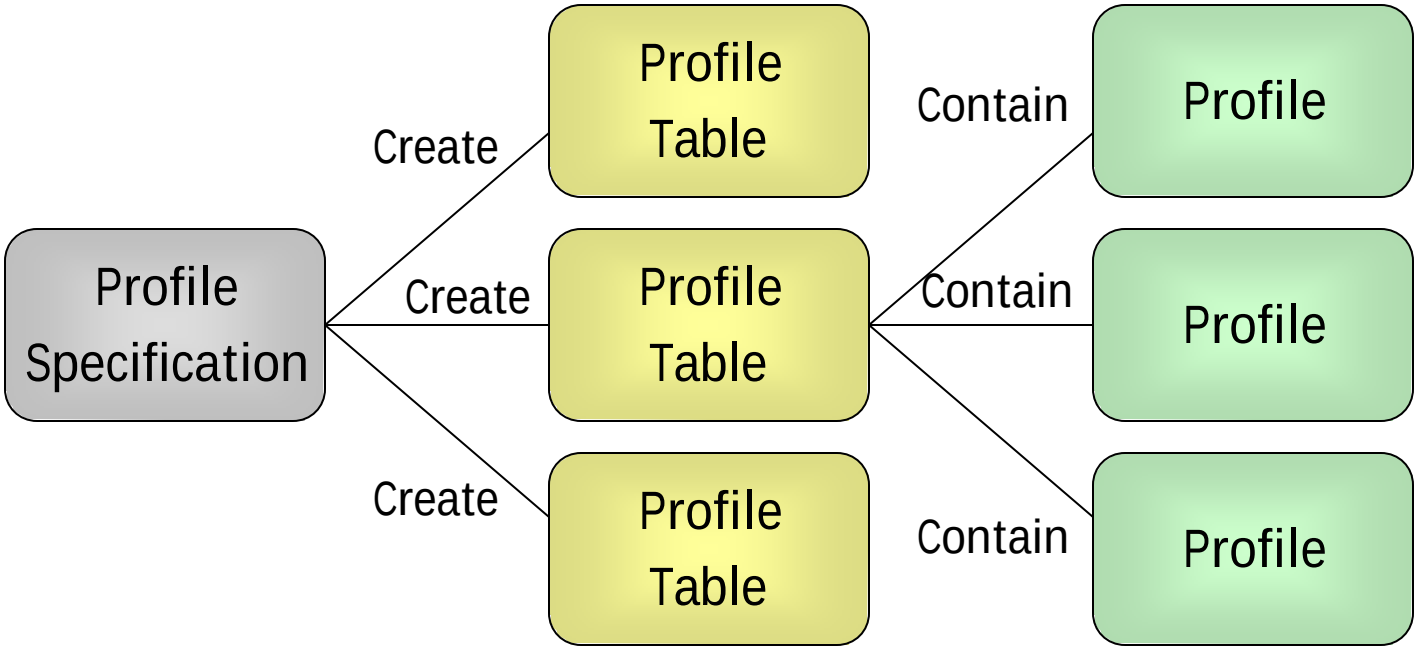
- The state of the SBB entity is the state of the CMP fields, in this case the CMP field counter.
  - Entity refers to this CMP state.

Counter SBB Example

- Different object instances can act on the same entity consider the following scenario:
  1. An SBB invokes operations:
     ```
     CounterSbbLocalObject counter = getCounterSbb();
     // using a CMP field as a local object
     counter.reset();
     counter.increment();
     ```
  2. The transaction commits
  3. The machine that step 1 was running on fails
  4. Another machines JVM invokes:
     ```
     CounterSbbLocalObject counter = getCounterSbb();
     // get an object for the same entity
     // counter.getValue() is still 1
     ```
- The SBB objects are not the same in 1 and 4 because they are in different JVMs, however the SBB state remains true across JVMs.

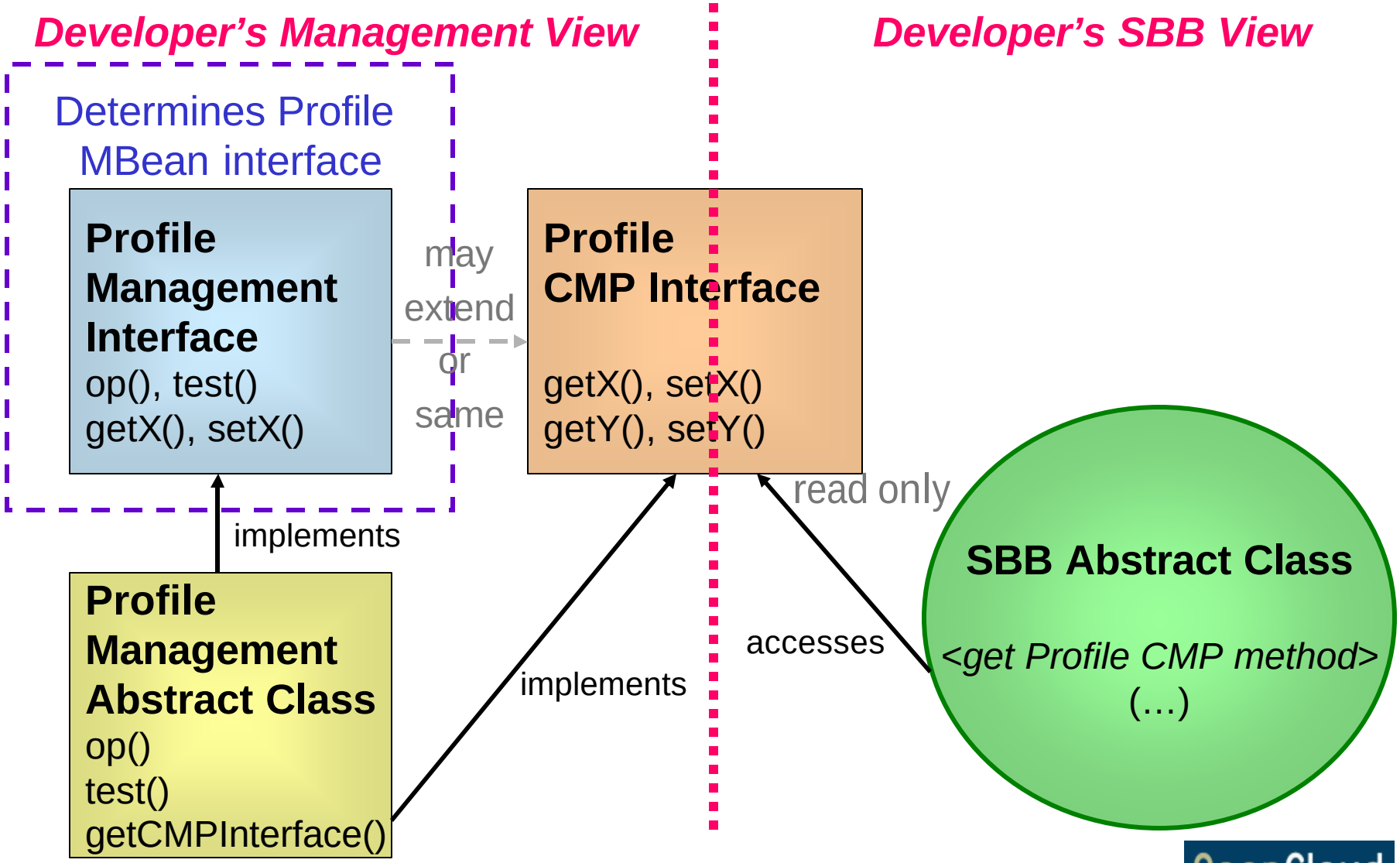
Profiles

Profile Concepts



- Profile Specification defines
  - Schema (i.e. attributes) of Profile Tables (and Profiles in Profile Tables)
  - Interfaces and classes of the Profile Specification
- Profile Tables are created from Profile Specification
- Profile Tables contain Profiles

Profile Specification



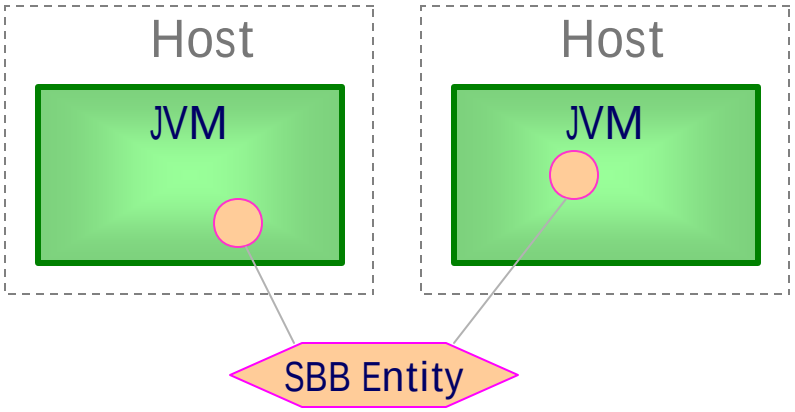
Transactions + Timers

Transactions

- Transactions are required primarily to simplify the job of the SBB developer.
  - SLEE uses transaction for ease of application development through a well-understood concurrency control and failure model.
- The key benefits the SLEE derives from transactions are
  - isolation for concurrency control
  - consistency when the JVM crashes in the middle of SBB code
- Transactions are a well defined and proven model
  - The infrastructure for implementing transactions (either for AC or SBB), will be reused to support transactions for each entity.
  - The complexity of supporting transactions beyond a single entity is the transaction co-ordination.
- Another use of transactions is demarcation
  - to replicate state to other nodes (for memory based redundancy)
  - to stable store (for disk based redundancy).

What happens if there are failures?

- State replication
  - Can access persistent state on another node
- Transactional semantics
  - Atomicity
    - All updates complete successfully or none completes
  - Isolation
    - Similar to some serial execution order
  - Automatic
    - No transaction API - (JTA not required)
    - Many implementations of transactional semantics possible
- Exception handling callbacks
  - Unchecked exceptions
  - Transaction rollbacks



Timers

- Timer uses native event model
- Timers may be late when
  - Container is overloaded
  - Container or container process has failed and restarted
  - Container explicitly stopped and restarted
- What to do when timers are late?
  - Preserve ALL
  - Preserve NONE
  - Preserve LAST
- Timer model tightly specified
  - Model includes clock and scheduler resolution
  - Pseudo code for model

Interaction + Integration

SLEE Feature Interaction

- The SLEE has various mechanisms built into its component model to help detect and resolve certain types of potential feature interactions
  - component model providers priorities
  - strong typing
  - activity context interface
  - per SBB state
- However the SLEE platform don't really understand the semantics of the application.

SLEE & EJB Integration

- EJB -> SLEE is an event submission interface
  - EJB submits events to the SLEE via the `SleeConnection` interface in the SLEE appendix
  - The inteface is implemented by a J2EE connector
  - This connector sends the events via some mechanism to a resource adaptor in the SLEE
  - The resource adaptor reads the events and passes them into the SLEE event processing sub-system
    - These events are processed like any other event

SLEE & EJB Integration

- SLEE -> EJB is an event submission interface
  - From EJB perspective an SBB is a remote client
  - An SBB may invoke an EJB though its home interface
  - EJB home interface is gained through the SBB's JNDI environment
  - The type of local interface is specified in the SBB's deployment descriptor

Resource Adaptors

- JAIN SLEE represents network resources as resource adaptors and each resource adaptor has a type
  - Resource adaptor type for JAIN SIP is ‘javax.sip’
- JAIN SLEE identifies Event by Event types
  - JAIN SIP Events are classified RequestEvents, ResponseEvents and TimeoutEvents, each of these classifications contains numerous types
  - For example the event type of a Request message of type ‘INVITE’ is ‘javax.sip.RequestEvent.Request.INVITE’
- JAIN SLEE represents the flow of events as activities
  - Activity Objects in JAIN SIP are ClientTransactions (locally initiated) and ServerTransactions (remotely initiated)

Presentation Outline

- Why create JAIN SLEE?
- Introduction to JAIN SLEE
- Implementation considerations

SLEE Performance Requirements

- Low latency
  - The *delay* for a packet of data to get from one designated point to another.
  - Time to setup a call < 500 ms, the average latency should between 50 to 100 ms range per transaction.
  - This should be achievable in a cluster of 2 to 4 nodes each with 2 to 4 CPUs with availability and redundancy enabled.
- High throughput
  - The ability to handle 1million Busy Hour Call Attempts (Operator specific)
  - This translates to approx 2500 transactions per second WITH state updates that are transacted and replicated to survive single node failures.
- High availability, 99.999% expected
  - < 6 minutes downtime per year (planned and unplanned)
  - Different notion of availability
  - Small partial failures (< 5%) count against availability
- Graceful handling of load spikes above maximum system capacity

Low Latency Requirements

What is required to set up a call in 500 ms?

- Traverse 2–5 network nodes or application servers
- Minimum 2 protocol messages events per node (application dependant)
- 1 transaction per event
- Approx 4 – 10 events to process within 500 ms
  (excluding propagation delay, some processing overlapped with propagation)
- 50 – 125 ms per transaction
  (for simple transactions with redundancy for single failure tolerance)
- 95th percentile round trip time is 200 ms

High Throughput Requirements

What does 1,000,000 BHCA require?

- 2 – 10 protocol messages or events per call (application dependant)
- 1 transaction per event
- Approx 280 call attempts per second
- 560 – 2800 transactions per second with transactional state updates and replication

SLEE Failures and Clean up

- If the SLEE fails before an event has been delivered to a particular SBB entity, it will be delivered on that SBB entity within another JVM or when the SLEE restarts.
  - If an SBB entity is invoked but has not completed processing the event, the SBB entity will be invoked again with a transaction rolled back callback.
  - This mechanism ensures the SBB entity knows that there is failure in the middle of processing an event
- All SBB entities are detached from an Activity Context upon receipt of an ActivityEndEvent
  - The key to cleaning up SBB entities is to ensure that all AC's have an end
  - A entire SBB tree will be cleaned up after all SBB entities in a SBB tree are no longer attached to any Activity Contexts.

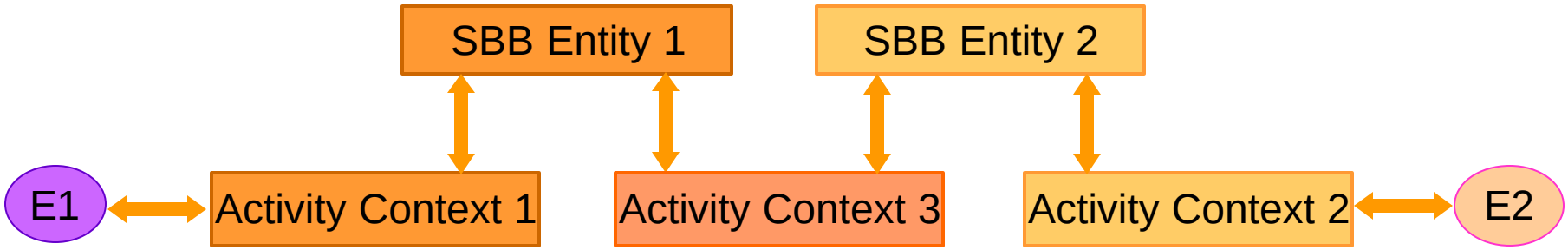
SLEE Concurrency Control

- There are three controls for concurrency in the SLEE

1. A single event is processing on an Activity at any time
   - An SBB object services a single event at a time, hence event delivery to a single SBB object is serialized.
2. Isolation of concurrent transactions, that is a transaction in process which is not yet committed must remain isolated from any other transaction.
   - There are some cases where an SBB entity may be concurrently invoked through multiple SBB objects that represent the same SBB entity.
     - The SBB objects servicing different events do not modify transactional state, they are read-only and can process events in parallel.
     - If optimistic concurrency control is chosen and the SBB objects modify the same transactional state one is required to abort.
3. SBB object will never receive a concurrent invocation.

SLEE Concurrency Control

- Example
  - 2 SBB entities both attached to two different Activity Contexts and same Activity Context.
  - SBB1 receives event E1 on AC1 in TX1 and SBB2 receives event E2 on AC2 on TX2.
  - Both SBB1 and SBB2 want to modify the transactional state of AC3.



- The SLEE needs to define concurrency control semantics for updates on AC3 by both SBB1 and SBB2.
  - TX1 and TX2 attempt to touch the same unit of transaction state, i.e. they both refer to AC3.
  - The isolation and semantics of access between the concurrent transactions will determine what happens, i.e. pessimistic or optimistic.

Design Implications

- Multi-site distributed system
- Main memory storage of application and infrastructure state
- Memory replication
- Overload management

Implementation Challenges

- Application logic frequently not idempotent
  - During failure recovery, some calls may be in unknown state
  - Acceptable to terminate calls
- More replicas with equivalent performance
  - More replicas reduce dropped calls on single failure
  - But more replicas should not reduce performance
- Lowering failure detection and recovery times
  - Reduce time that calls are assigned to a failed node
  - Reduce calls in unknown state
- Integration with external systems
  - Different notions of failure
  - Service availability depends on whole system

Java Specific Challenges

- Node re-starts
  - JVM boot up time
  - JIT takes time to take effect
  - Work around by having each process run through iterations of the critical path before joining the cluster
  - More than 2 cluster members helps reduce impact of long boot times
- Garbage Collection
  - Impacts 95th percentile latency
  - Impact absolute time for failure detection and cluster membership voting algorithms
  - Impact reduced significantly since Parallel new GC and CMS collectors introduced

Conclusion

JAIN SLEE is a generic application server designed specifically for high performing event-driven applications

SLEE in Communication Networks

- 2G and 2.5G Networks
  - Service Control Point
  - Service Node (2G to 3G using a Media Gateway)
- 3G Networks
  - Service Switching Control Point
  - Service Switching Point
  - SIP Proxy
  - 3GPP IMS CSCFs
- Convergent Networks
  - Gatekeeper
  - Common Service Delivery Platform
  - Convergent SCP, SSCP, SSP
  - OSA Gateway

Other technologies and SLEE

- Auto-ID
- Manufacturing
- Real-time health monitoring
- Industrial flow control
- ……

JSR 22

http://jcp.org/en/jsr/detail?id=22

Subscribe to:

http://archives.java.sun.com/jain-slee-interest.html