

Cyber-entities

Jan. 13, 2003

Jun Suzuki, Michael Le and Tatsuya Suda

{jsuzuki, suda}@ics.uci.edu

mvl@uci.edu

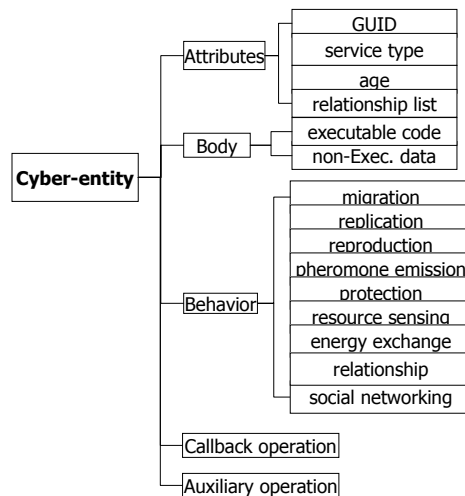
<http://netresearch.ics.uci.edu/bionet/>

Dept. of Information and Computer Science
University of California, Irvine

1

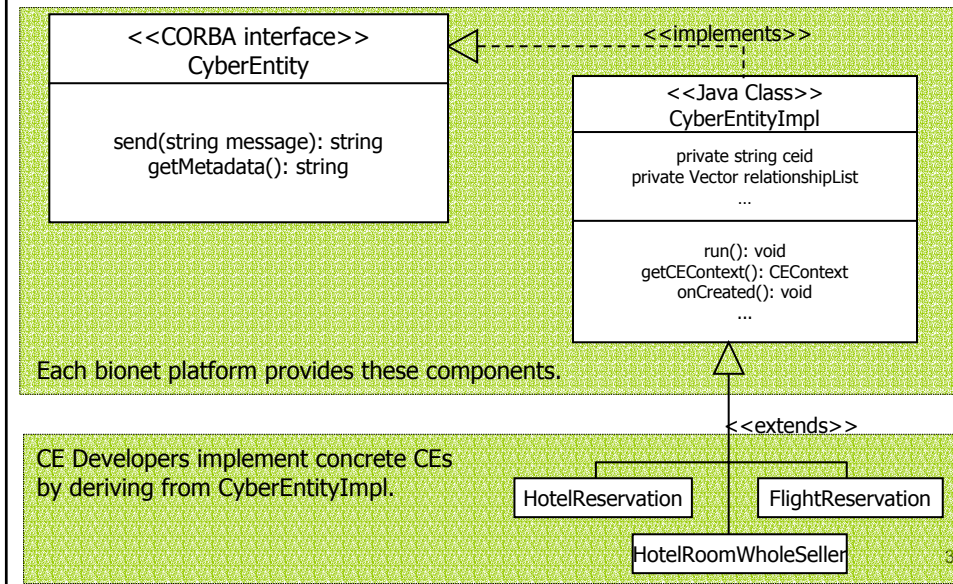
Cyber-entity (CE)

- Cyber-entity is the smallest component in bionet environment.



2

CE's Base Class/Interface Structure



CyberEntity

- `CyberEntity` is the root interface for all the CEs .
 - specified with CORBA IDL (Interface Definition Language).
 - defines the operations that can be called by other CEs.
 - `interface CyberEntity`

```

{
    oneway send(in string message);
    string metadata();
};

```
 - A subset of FIPA ACL with some extensions (called bionet communication language, or BCL) is used as a communication language.
 - encoded with XML

CyberEntityImpl

- `CyberEntityImpl` is the base Java class for all the CEs.
 - Every CE derives from this class.
 - defines a set of operations and attributes that are common among CEs.
 - provided by all the bionet platforms.

5

CE's Attributes

- CE's attributes are maintained in instance variables of `CyberEntityImpl` and its descendant classes.
- A CE can know another CE's attributes by obtaining the CE's metadata.
 - through `metadata()`, defined in `CyberEntity`.

6

CE's Behaviors

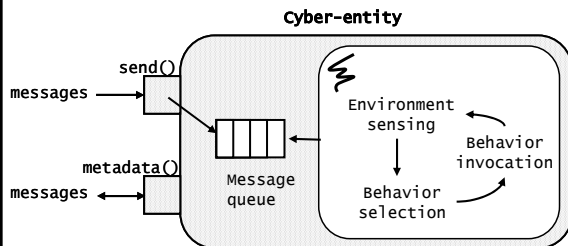
- Each CE can have 10 behaviors.
- CEs invokes behaviors by using corresponding bionet services.
 - Energy exchange and storage
 - Bionet energy management service
 - Migration
 - Bionet migration service
 - Replication
 - Bionet lifecycle service
 - Reproduction
 - Bionet lifecycle service
 - Death
 - Bionet energy management service

7

- Relationship establishment/maintenance
 - Bionet relationship management service
- Discovery of other CEs
 - Bionet social networking service
 - Bionet CE sensing service
 - Bionet pheromone emission service
- Resource sensing
 - Bionet resource sensing service
 - Bionet topology sensing service
- State change
 - Bionet lifecycle service
- Communication
 - through sending/receiving BCL messages

8

CyberEntity Structure



- Each CE uses an individual thread to continuously
 - sense the nearby environment,
 - identify behaviors suitable for the current environment condition, and
 - invoke the most suitable behavior

9

CE's Callback Operations

- Each CE has 10 callback operations named "on***".
 - They are defined in the CyberEntityImpl interface.
 - Implementation of each callback operation is left to CE developers.
 - The bionet platform provides examples and suggestions on how to implement each callback operation, but they are not mandatory requirements.
- Each callback operation is implemented by CE developers and then invoked automatically by bionet lifecycle service or bionet migration service when a specific event occurs.
 - The default implementation of each callback operation is empty (doing nothing).

10

- `public void onCreate(long ceid,
 Object ref)`
- `public void onReplicated(long ceid,
 Object ref,
 Object parentRef)`
- `public void onReproduced(long ceid,
 Object ref,
 Object parentRef1,
 Object parentRef2)`
- `public void onMove()`
- `public void onArrival(long ceid,
 Object reference)`
- `public CyberEntityImpl onActivated()`
- `public void onDeactivated()`
- `public void onDestroyed()`
- `public void onAutomated()`
- `public void onAutonomousToActivated()`

11

- (1) `onCreated()` is called when (after) a CE is created.
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to
 - create the CE's metadata
 - register the CE to a local directory
 - initialize the CE's behavior selection engine.
- (2) `onReplicated()` is called after a CE is replicated.
 - This is called on a replicated child CE by a bionet lifecycle service.
 - A typical implementation of this operation is to invoke the CE's internal "replicated" state.
 - The "replicated" state may
 - » inherit a relationship list that a parent CE has (Mutation on a relationship list may occur at this point).
 - register the CE to a local directory.
 - establish a relationship between child and parent CEs.

12

- (3) onReproduced() is called after a CE is reproduced.
 - This is called on a reproduced child CE by a bionet lifecycle service.
 - A typical implementation of this operation is to invoke the CE's internal "reproduced" state.
 - The "reproduced" state may
 - » inherit and crossover relationship lists that a parent CEs have (Mutation on relationship list may occur at this point).
 - » register the CE to a local directory.
 - » establish relationships between child CE and parent CEs.
- (4) onMove() is called when (before) a CE migrates to another host.
 - called by a bionet migration service
 - This operation may invoke the CE's internal "emitPheromone" state.
 - The "emitPheromone" state emits the CE's pheromone.

13

- (5) onArrival() is called when (after) a CE arrives from another host.
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to invoke the CE's internal "arrive" state.
 - The "arrive" state may register the CE to a local directory, and update its bi-directional relationships.
- (6) onActivated() is called when (after) a CE is activated.
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to invoke the CE's internal "activated" state.
 - The "activated" state may restore the CE's state.

14

- (7) onDeactivated() is called when (before) a CE is deactivated.
 - called by a bionet lifecycle service
 - This operation, for example, externalizes the CE's state (e.g. in a file or database).
- (8) onDestroyed() is called when (before) a CE is destroyed.
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to invoke the CE's internal "destroyed" state.
 - The "destroyed" state may free resources the CE has used and remove the CE's reference from a local bionet directory service.

15

- (9) onAutomated() is called when (before) a CE is automated.
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to invoke the CE's internal "autonomous" state.
 - The "autonomous" state may allow CE's to autonomously sense environment, process requests, and perform behaviors.
- (10) onAutonomousToActive() is called when (before) a CE goes from being Autonomous to Active
 - called by a bionet lifecycle service
 - A typical implementation of this operation is to invoke the CE's internal "active" state.
 - The "active" state may allow CEs to stop processing incoming requests and thus conserve energy

16

- Different CEs have their own implementations of each callback operation.
 - Different CEs can perform differently even when the same event occurs.
 - e.g. When a CE with its GUI presentation arrives from another host, the CE's onArrive() may check if any GUI toolkit is available in the local host and the available toolkit is compatible with the CE and then build the CE's GUI presentation.
 - e.g. When a CE is deactivated, the CE's onDeactivated() may store its state in a file, or may not store its state.
 - The difference between a deactivated and destroyed CE is described later.

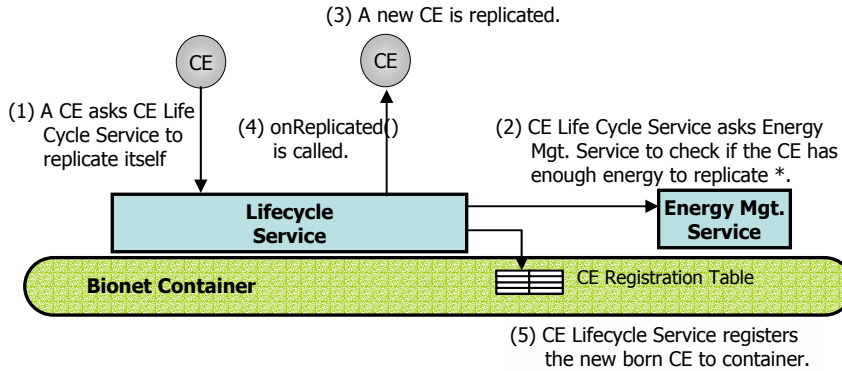
17

- The implementation of a callback operation can be empty.
 - A CE's developer may not implement a callback operation.
 - Every callback operation is invoked by Bionet Services when a corresponding event occurs. However, CE may not perform anything.
- Bionet services should invoke CE's callback operations because CE sometimes does not know when and which event occurs.
 - e.g. A reproduced CE does not know when the reproduction completes.
 - Bioent Life Cycle Service knows this timing. So, the service should invoke corresponding callback operations (i.e. onCreated() and onActivated()) on the replicated CE.

18

How Callback Ops are Called?

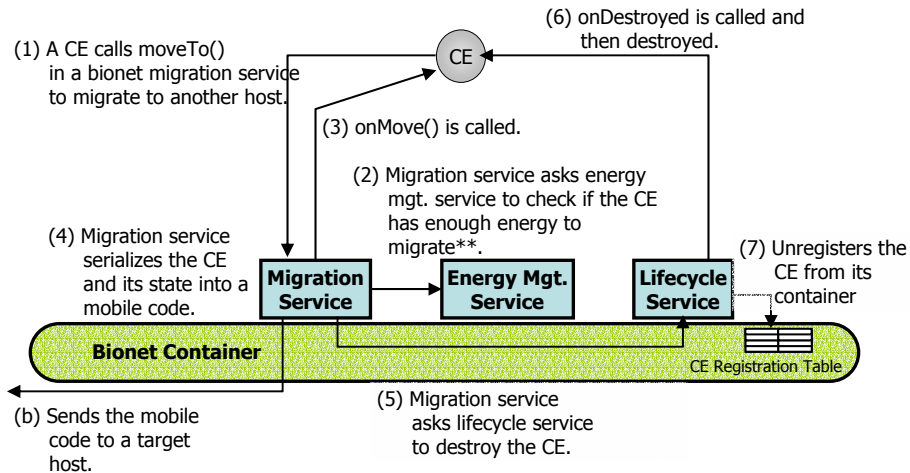
- There are 2 types of sequences to invoke callback operations.
 - (1) A CE calls an underlying bionet service, and then the service invokes an appropriate callback operation.



* Bionet Life Cycle may not check the replicating CE's energy level; It may trust the CE. Please see also the section on bionet lifecycle service for details.

19

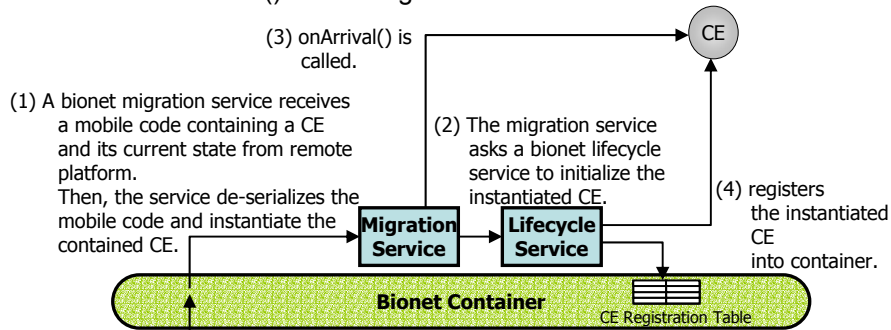
– another example



** A bionet lifecycle service may not check the replicating CE's energy level; trust the CE. Please see also the section on bionet lifecycle service for details.

20

- (2) A container calls a bionet service and then the service invokes an appropriate callback operation.
 - This case applies only when a CE arrives from another platform. A container that receives the CE's mobile code calls a bionet migration service. Then, the service invokes onArrive() on the migrated CE.



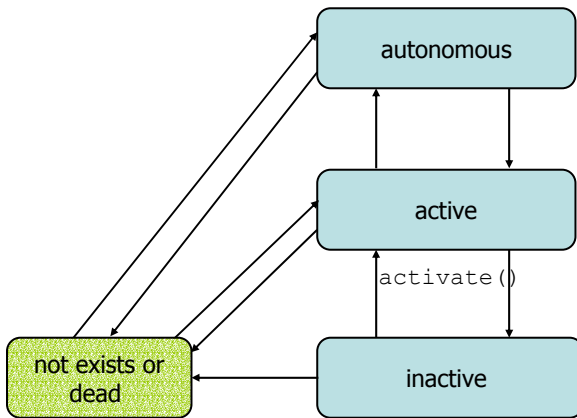
21

Auxiliary Operations

- Auxiliary operations are all the operations that do not belong to body, behavior and callback operations.
- They are designed as public and private methods.
 - CE itself, local bionet services or local bionet containers can call them.
 - `private CEContext getCEContext();`
 - obtains a CE context associated to a CE
 - `public void setCEContext(CEContext ctx);`
 - associates a CE context to a CE
 - A bionet life cycle service calls this operation.
 - `public int getCEState();`
 - gets and return the associated CE state
 - `private void log();`
 - logs a CE's status

22

CE's State Transition



Each CE may have 4 states during its lifetime.

It changes its state voluntarily; any CEs are not allowed to change another CE's state.

23

- An autonomous CE
 - receives messages from other cyber-entities.
 - continuously senses nearby environment and performs its behaviors.
 - runs on an individual thread; thus, expends energy continuously for using a thread (CPU) and memory space.
- An active CE
 - is ready for receiving messages from other cyber-entities, but
 - does not perform behaviors
 - consumes memory; thus expends energy continuously for using memory space.
- An inactive CE
 - is in “sleeping” state in which it is externalized into a file
 - does not expend energy because it does not consume any resources.

24

CyberEntityState

A Java class that defines all the different types of available CE states. Currently there are 4 states a CE can be in.

- Contains public static final data fields:

```
int CyberEntityState.AUTONOMOUS = 0;
```

```
int CyberEntityState.ACTIVE = 1;
```

```
int CyberEntityState.INACTIVE = 2;
```

```
int CyberEntityState.DESTROYED = 3;
```

- CEs can use this class to specify the different states that the CEs are in.

25

CEActivator

Bionet Platform's servant (CE) manager.

A CORBA Object that provides the capabilities to incarnate and etherealize CEs.

```
package edu.uci.ics.bionet.ce;
public class CEActivator extends _ServantActivatorLocalBase
{
    public CEActivator()
    public org.omg.PortableServer.Servant incarnate(byte [] oid, POA bioContainer) throws
    org.omg.CORBA.OBJECT_NOT_EXIST
    public void etherealize(byte [] oid, POA bioContainer, Servant ce, boolean cleanup_in_progress,
    boolean remaining_activations)
}
}
```

- When a message arrives to an inactive CE, POA calls incarnate() on the CEActivator, which in turn calls activate() in the LifeCycleService to activate the CE.
- When a CE wants to deactivate or destroy itself, CEActivator's etherealize will get called to remove the CE from the Bio-Container.
 - CEActivator will be invoked automatically by the Bio-Container when the appropriate function is needed

26