

The JADE services architecture



Giovanni Caire

Telecom Italia

giovanni.caire@tilab.com

Ideas and motivations

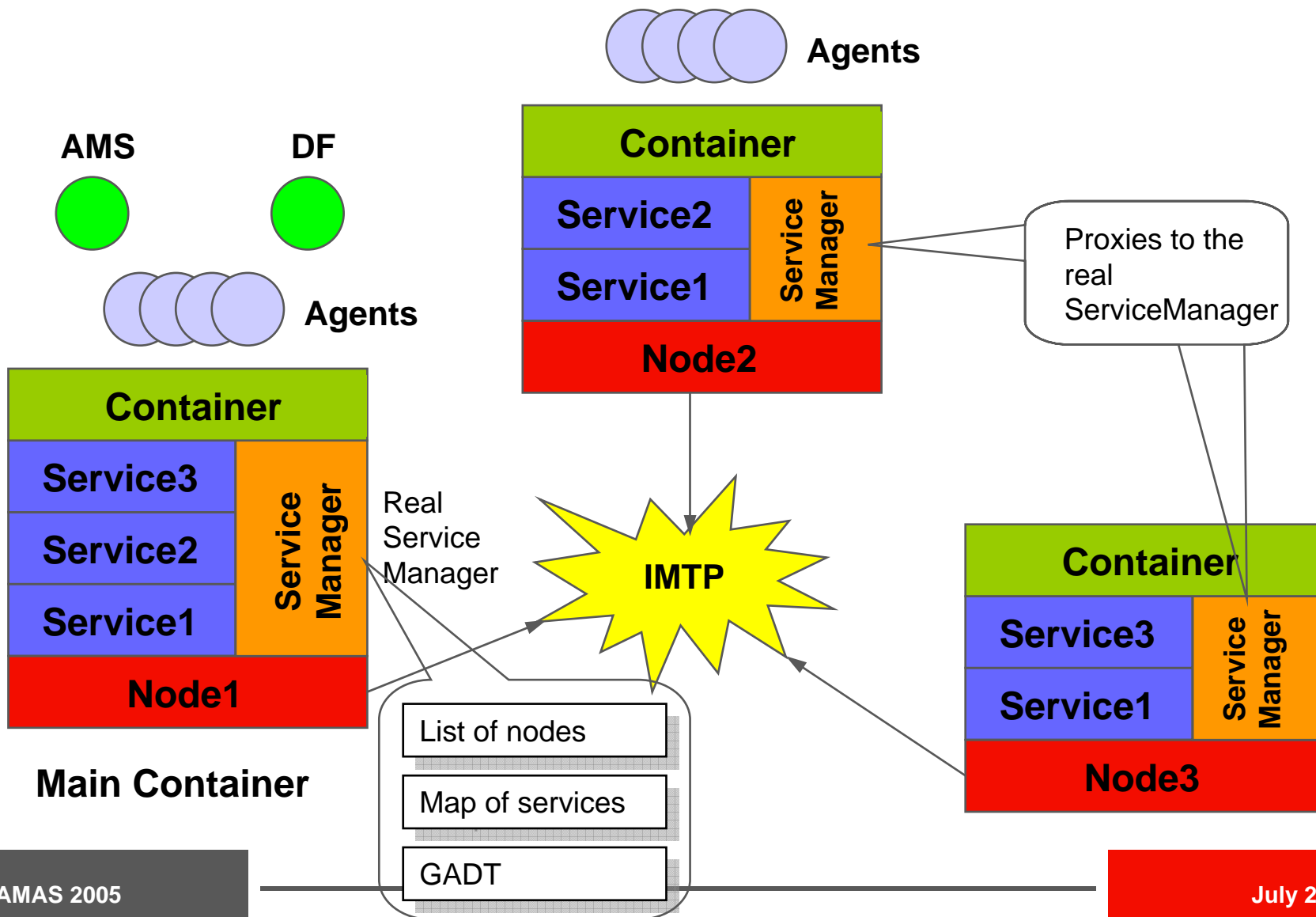
Requirements

- 1) Fine grained granularity of platform features
- 2) Open-ended set of features
- 3) Distribution
- 4) Flexible deployment strategy to target the hybrid wireline/wireless environment

Main abstractions

- Node
- Service

Architecture overview



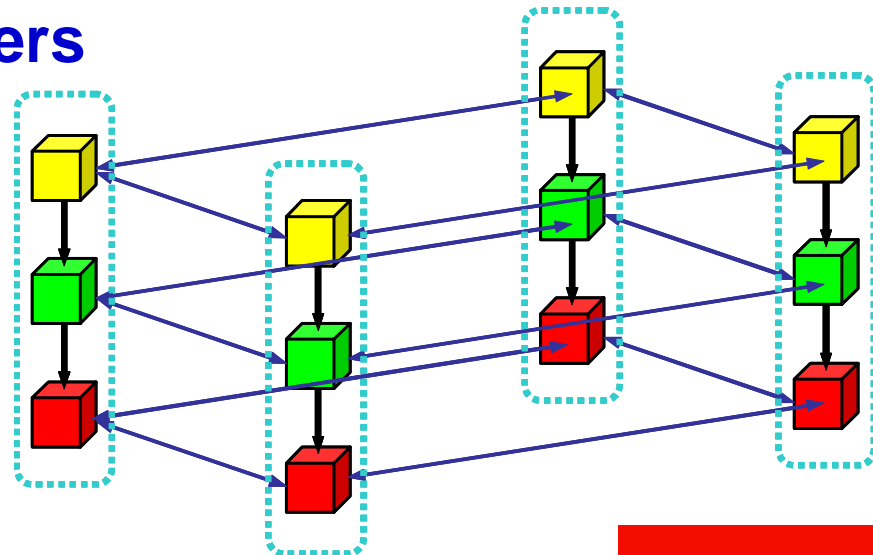
The Distributed coordinated filters

- **Inspiration from Aspect Oriented Programming**

- Separation of concerns + Aspect Weaving
- Composition Filter approach: Each object is provided with
 - An incoming filter chain whose filters are invoked whenever the object receives a method call
 - An outgoing filter chain whose filters are invoked when the object is about to call another object's method

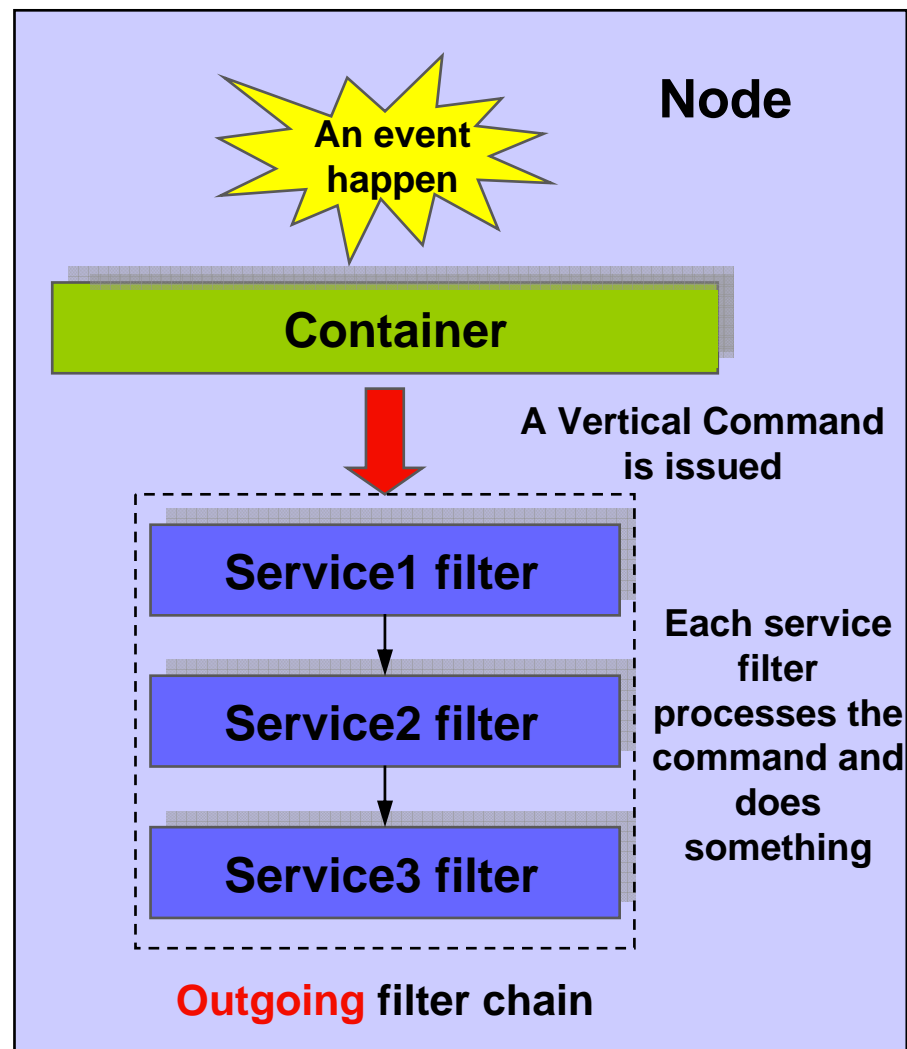
- **Distributed coordinated filters**

- Aspect => Service
- Object => Node
- Each Service is “sliced” over the nodes



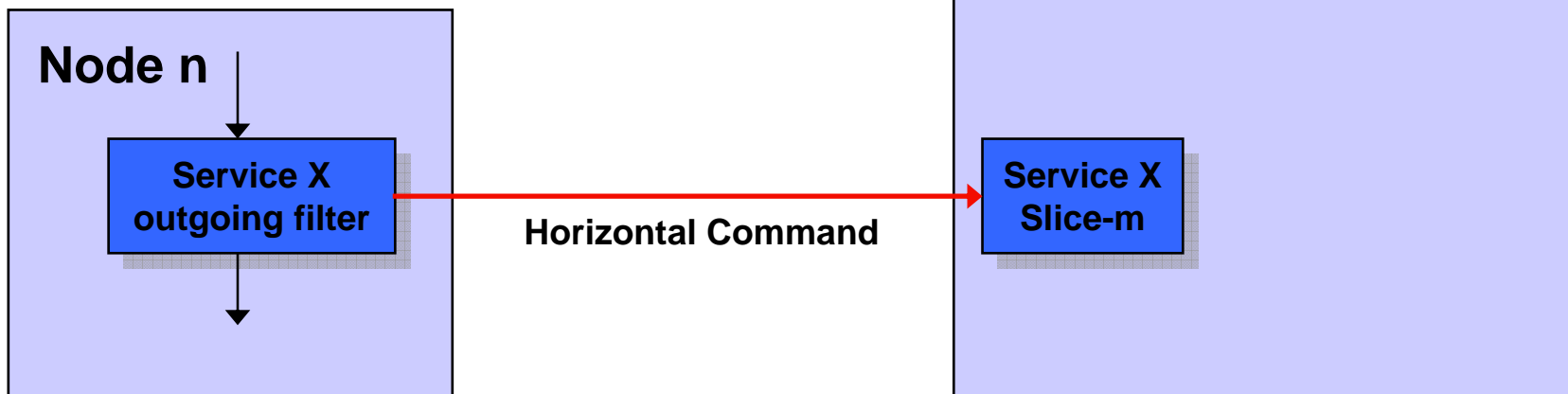
The Outgoing filter chain

- All events that happen at the agent level trigger a **Vertical Commands**
- Each Service may provide an **Outgoing Filter** and all Vertical Commands are processed sequentially by the filters of all services installed in the local node.
- Each filter can act on certain Vertical commands and ignore the others



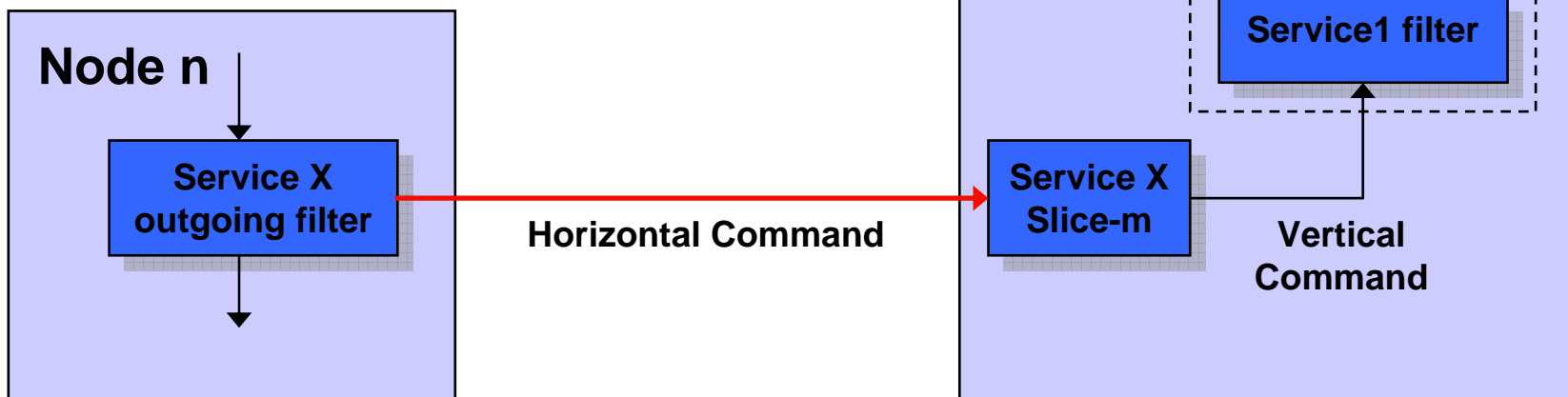
Horizontal Commands and Slices

- When processing a Vertical Command a filter on a given node may need to interact with the “slices” of its service on other nodes
- These interactions are carried out by means of **Horizontal Commands**
- Each service that requires node-to-node interactions must provide a **Slice** to serve Horizontal Commands

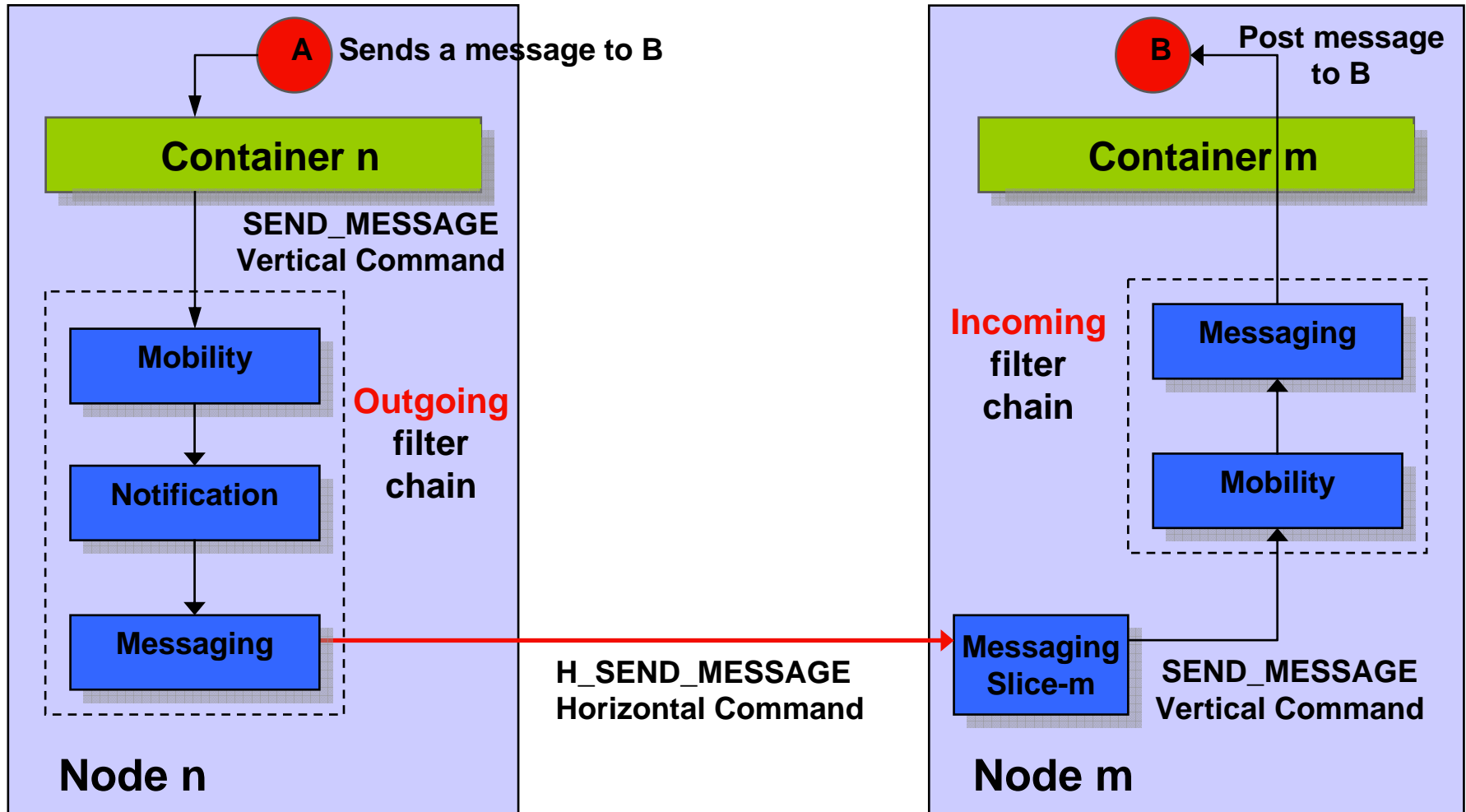


The Incoming filter chain

- When a Slice serves an HorizontalCommand it may trigger a new Vertical Command
- Each Service may provide an **Incoming Filter** and all Vertical Commands issued by Slices are processed sequentially by the filters of all services installed in the local node



An example



How an agent can interact with a service

- **Some services may be directly accessed by agents through a **ServiceHelper****
- **Service helpers can be retrieved by means of the `getHelper()` method of the `Agent` class**
- **E.g.**
 - `SecurityHelper sh = (SecurityHelper) getHelper("security");`
- **For backward compatibility reasons some services are not accessed by means of a `ServiceHelper`, but by means of methods of the `Agent` class.**
- **E.g.**
 - `Agent.send()` instead of `MessagingHelper.send()`
 - `Agent.doMove()` instead of `MobilityHelper.doMove()`

Sample code: The SniffingService

```
public class SniffingService extends BaseService {
    private Filter myFilter = new SniffingFilter();
    private ServiceHelper myHelper = new SniffingHelperImpl();

    /**
     Retrieve the filters of this Service
    */
    public Filter getCommandFilter(boolean direction) {
        if (direction == Filter.OUTGOING) {
            return myFilter;
        }
        else {
            // We are only interested in sent messages → No incoming Filter
            return null;
        }
    }

    /**
     Retrieve the helper of this Service
    */
    public ServiceHelper getHelper(Agent a) {
        return myHelper;
    }
    .....
}
```

Sample code: The SniffingFilter

```
private MessageTemplate myTemplate;

/**
  Inner class SniffingFilter
  The filter that actually sniffs messages.
 */
private class SniffingFilter extends Filter {
    public boolean accept(VerticalCommand vc) {
        if (vc.getName().equals(MessagingService.SEND_MESSAGE)) {
            Object[] params = vc.getParams();
            AID sender = (AID) params[0];
            GenericMessage gMsg = (GenericMessage) params[1];
            ACLMessage msg = gMsg.getACLMessage();
            if (myTemplate != null && myTemplate.match(msg)) {
                System.out.println("Matching message");
                System.out.println(msg);
            }
        }
        return true;
    }
} // END of inner class SniffingFilter
```

Sample code: The SniffingHelper

```
/**  
  Inner class SniffingHelperImpl  
  Allows agents to set templates for messages to be sniffed  
  */  
private class SniffingHelperImpl implements SniffingHelper {  
    public void init(Agent a) {  
    }  
  
    public void setTemplate(MessageTemplate tpl) {  
        myTemplate = tpl;  
    }  
} // END of inner class SniffingHelperImpl
```

.....

```
public interface SniffingHelper extends ServiceHelper {  
    public void setTemplate(MessageTemplate tpl);  
}
```

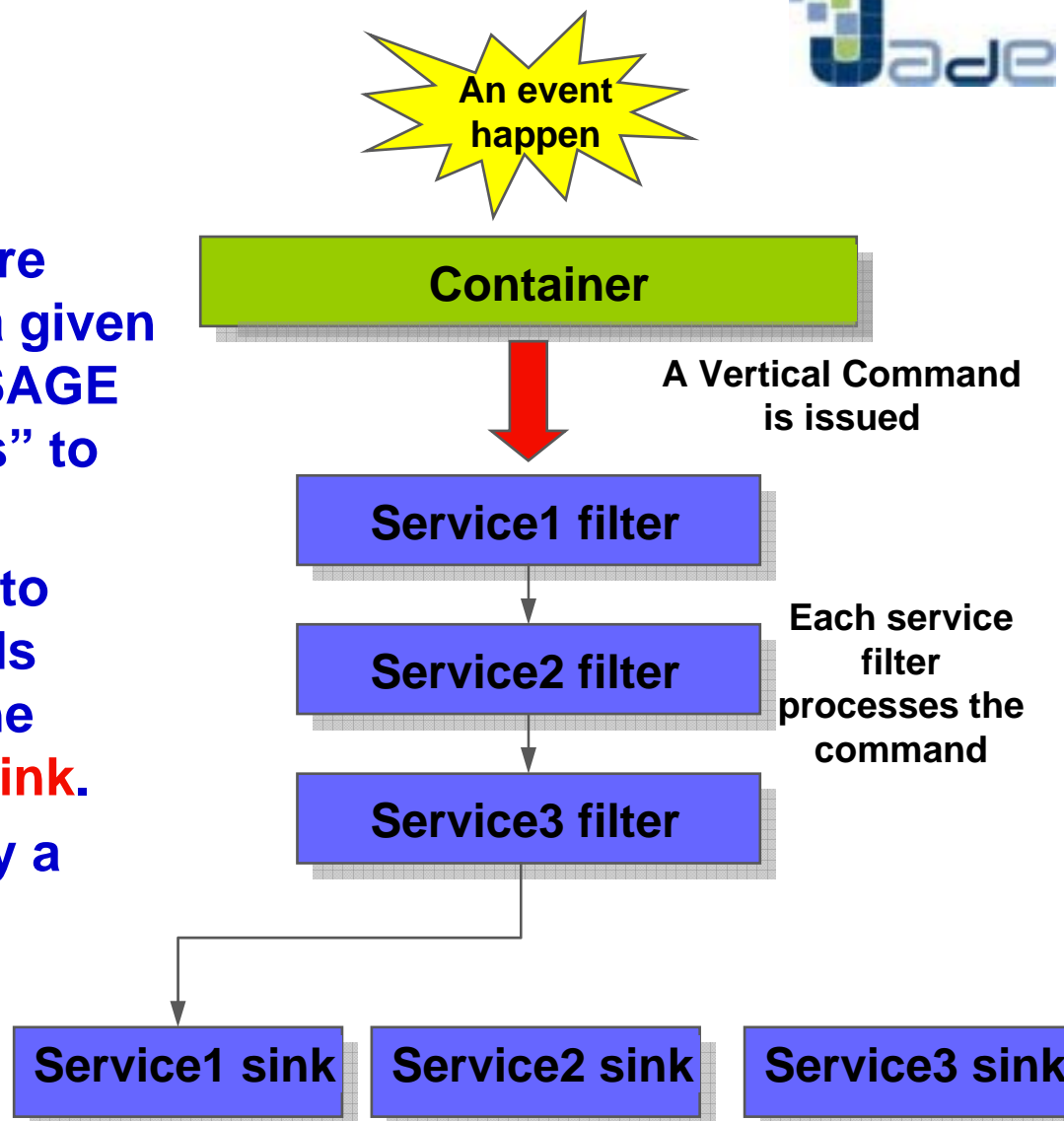
Sample code: Accessing the helper



```
.....  
try {  
    SniffingHelper myHelper = (SniffingHelper) getHelper(SniffingService.NAME);  
    myHelper.setTemplate(MessageTemplate.MatchPerformative(ACLMessage.REQUEST  
    ));  
}  
catch (ServiceException se) {  
    se.printStackTrace();  
}  
.....
```

Sinks

- Some vertical commands are intrinsically associated to a given service (e.g. a SEND_MESSAGE vertical command “belongs” to the Messaging Service)
- Each service has a chance to consume its own commands after they have traversed the filter chain by means of a Sink.
- Each service may have only a filter, only a sink or both



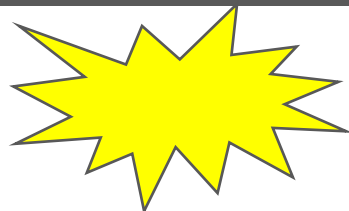
A Vertical Command is issued

Each service filter processes the command

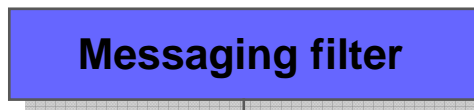
Only the sink of the service that owns the command consumes it

An example

An Agent sends a confidential message



A SEND_MESSAGE Vertical Command (embedding the message as a parameter) is issued



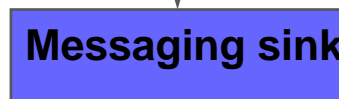
The Messaging filter encodes the payload



The Event-notification filter notifies the Sniffer



The Encryption filter encrypts the payload

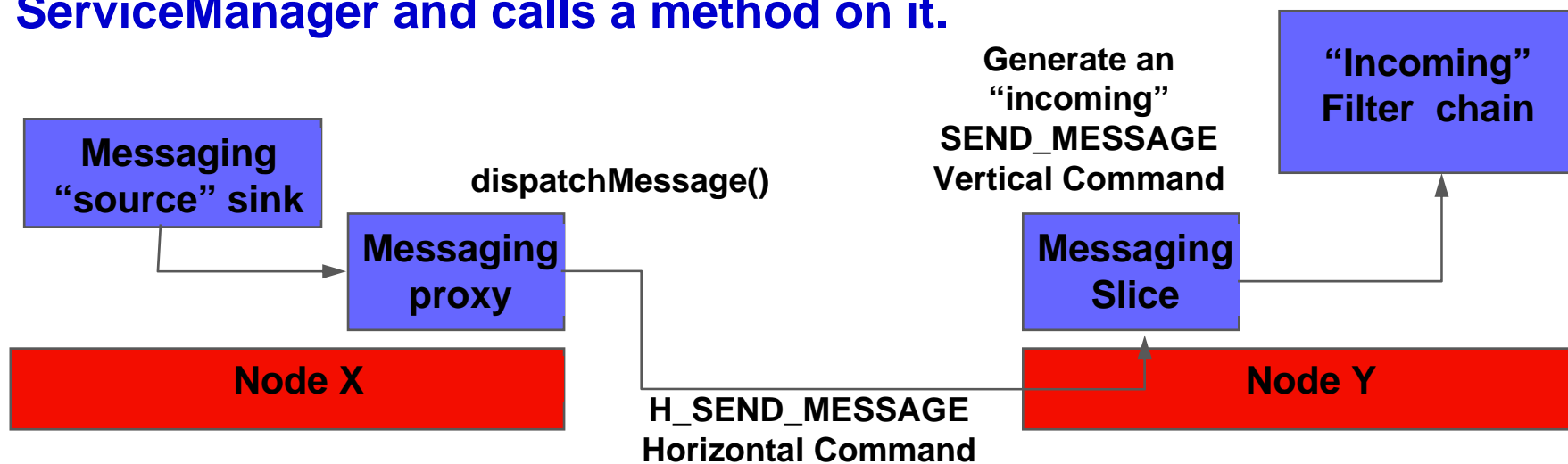


The Messaging sink dispatches the message to the receiver

Sinks of other services

SliceProxies

- Are in charge to transform method calls to remote slices into Horizontal Commands
- Implement the Horizontal Interface of a service (method `getHorizontalInterface()`)
- The name must be `<Service-name>Proxy`
- When a service filter or sink needs to interact with a slice on a remote node it retrieves a proxy to that slice through the `ServiceManager` and calls a method on it.



Summary

- **A Service is composed of**
 - Outgoing & incoming filters (for processing Vertical Commands)
 - Source & target sinks (for consuming owned Vertical Commands)
 - Slice and SliceProxy (for node-to-node interactions)
 - Helper (for agent interactions)
- **Mostly all JADE features are currently implemented as services**
- **People interested in modifying/extending JADE features should consider the development of new services as the first option**

Thanks for your attention

• Questions?