AOSI

Programming Abstractions for Distributed Computing

Programming Abstractions for Distributed Computing

Why ?

Motivating Example:

Data: Shared Variable
Operation: Read/Write

Is this the right granularity of data and ops?

Goals:

- **Independent development and evolution**
- **Well-defined interfaces and information hiding**
- **Larger entities for deployment and distribution**
- **Well-defined complex operations**

Programming Abstractions for Distributed Computing

How? Discussion:

	objective	granularity	benefit
• Objects	information hiding	fine	design
• Components	deployment	fine	config
• Services	independance, dynamic	coarse	runtime
• Actors	object+explicit_sync	medium	design

Programming models: Object

Object: Incarnation of an abstract data type.
Characteristics of o-o: class, inheritance, polymorphism

- An object is a unit of instantiation; it has a unique identity.
- An object has state; this state can be persistent state.
- An object encapsulates its state and behaviour.

Programming models: Component

Component: encapsulated unit of functionality and deployment that interact with other components only via well defined interfaces.

- ***Interfaces:*** **defining sets of operations and the associated data types.**
- ***Receptacles:*** **special (required) interfaces that explicitly define the dependencies on other components. On deployment this describes which other components must be present.**
- ***Binding:*** **association between one single interface and one single receptacle.**
- ***Capsule:*** **container providing the run-time API, e.g. a process**

Programming models: Component

Clemens Szyperski, Component Software, ACM Press/ Addison-Wesley, England, (1998).

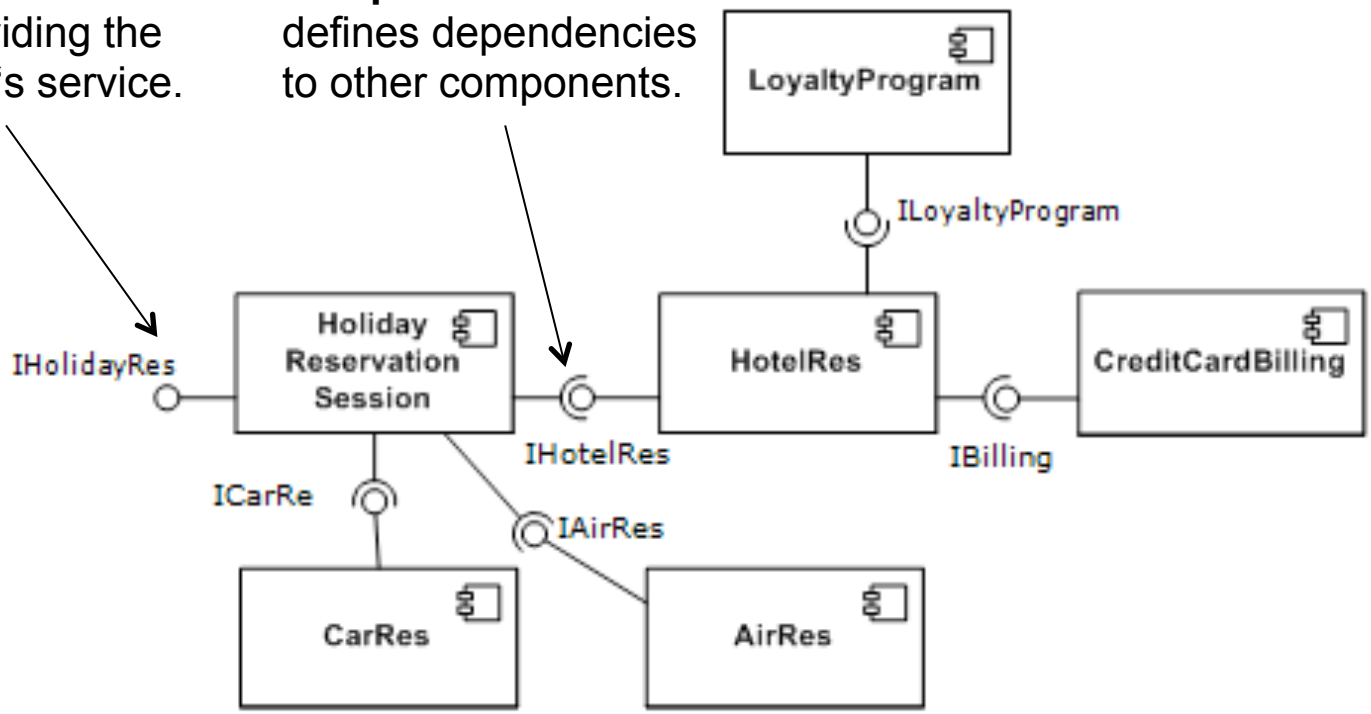
A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

A component is a unit of independent deployment.
A component is a unit of third-party composition.
A component has no persistent state.

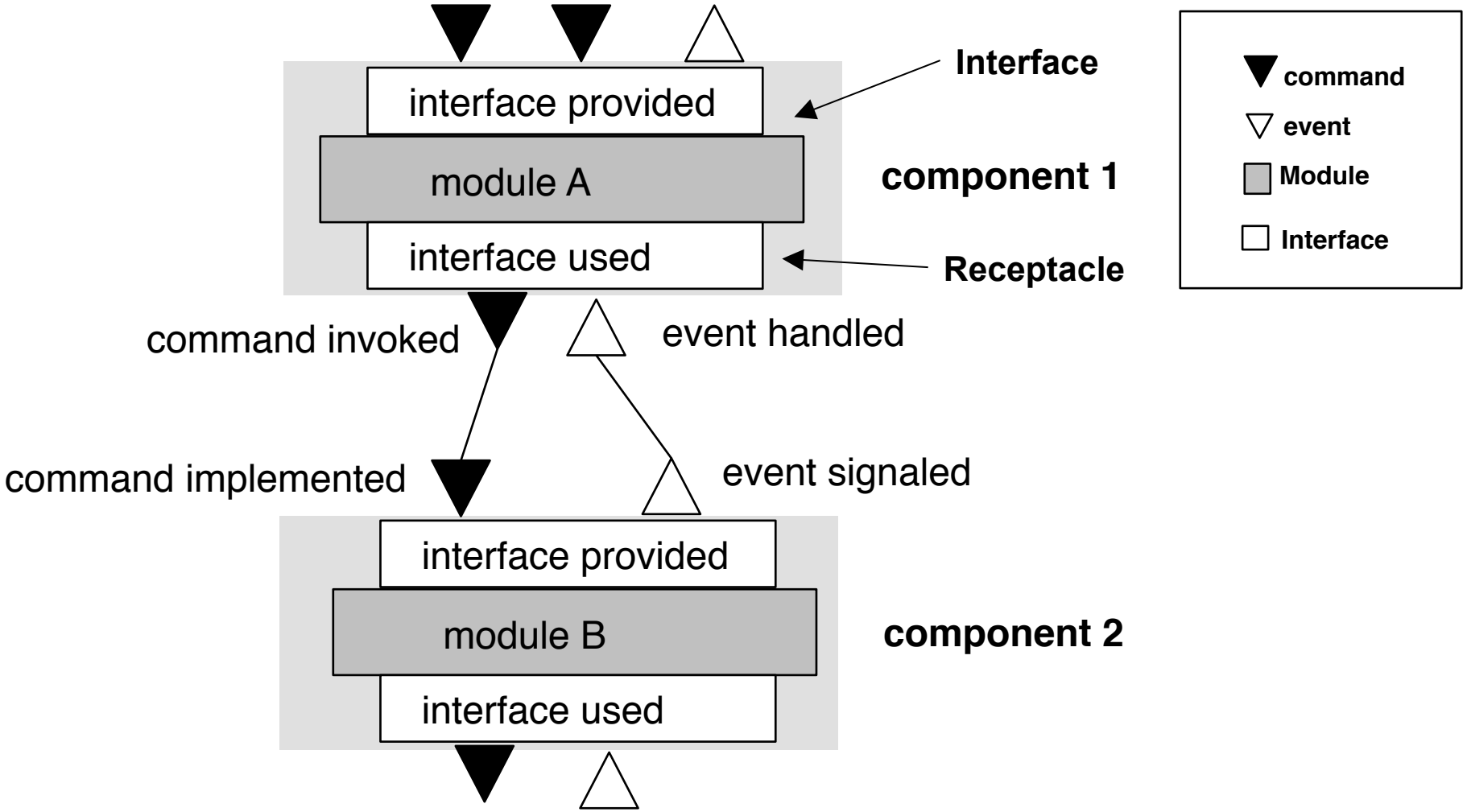
Programming models: Component

interface.
way of providing the component‘s service.

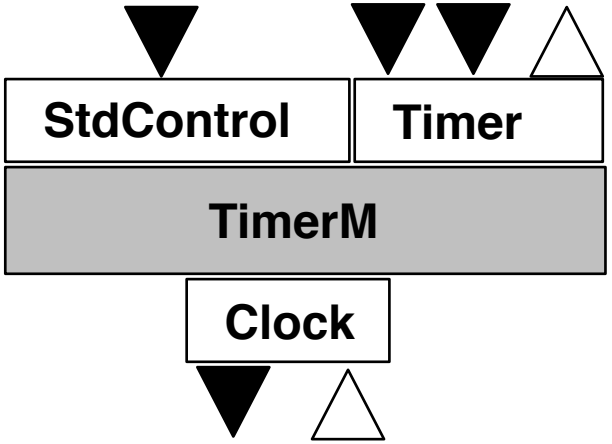
receptacle:
defines dependencies to other components.



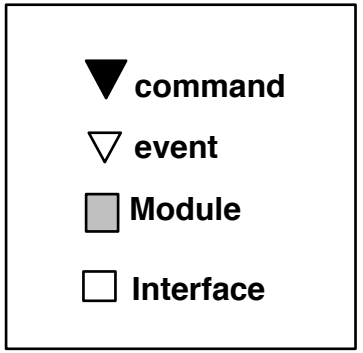
Programming models: TinyOS Component Model



Programming models: TinyOS Component Model

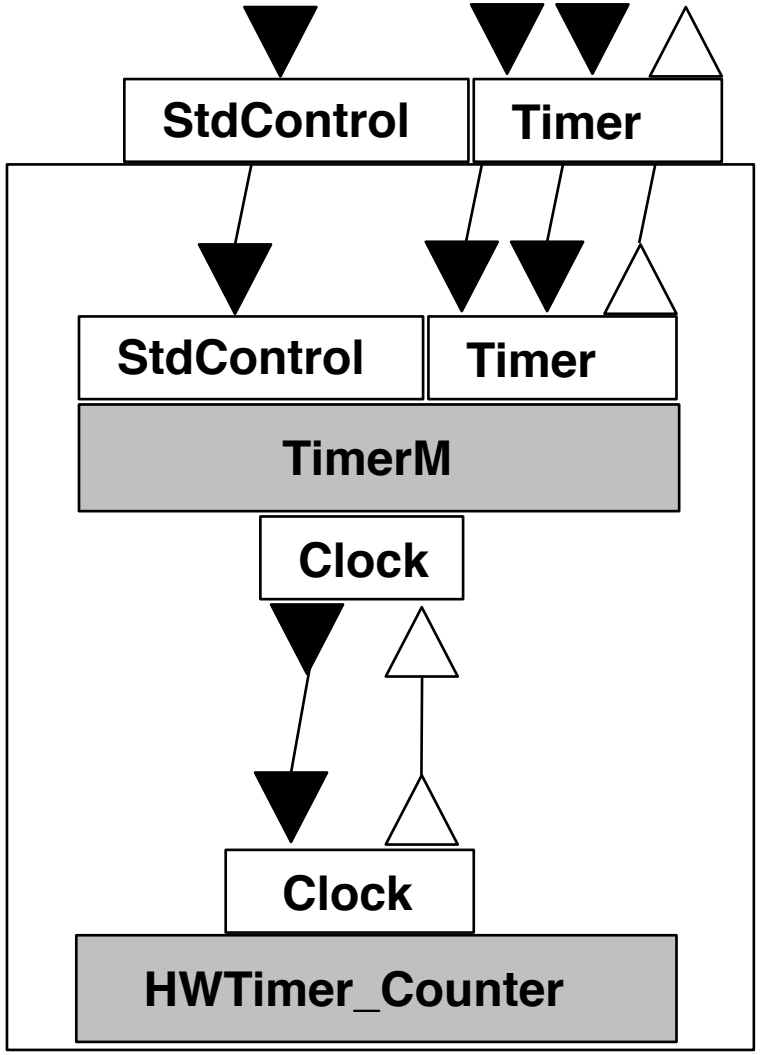



```
Module TimerM {
  provides {
            interface StdControl
            interface Timer;
  }
use interface Clock as Clock;
} . . .
```




```
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

Programming models: TinyOS Component Model



TinyOS Configuration:

```
configuration TimerC {
   provides {
       interface StdControl;
      interface Timer;
   }
}


implementation {
   components TimerM, HWTimer_Counter;
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;
   TimerM.Clk → HWTimer_Counter.Clock
}
```

Programming models: Service

What is a service?

Service:

"a mechanism to enable access to one or more capablities, where access is provided using a prescribed interface and is exercised consistently with the constraints and policies as specified by the service description." (OASIS)

"a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format." (W3C)

Properties of a service

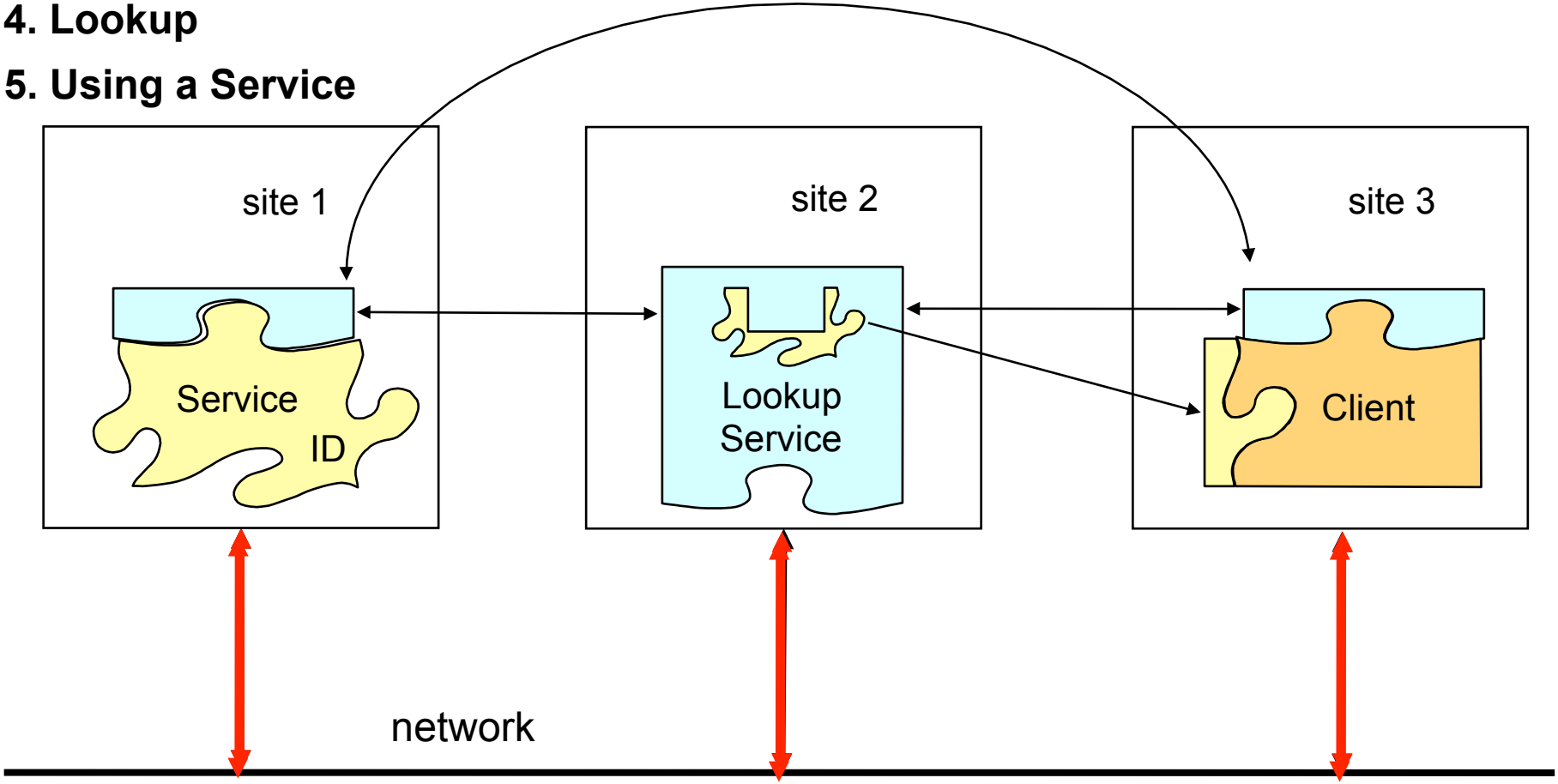
- A service can be used as an independent and self-contained entity.
- A service is available within a network.
- Every service has a published interface that is sufficient to use the service.
- The use of services is platform and language independent.
- A service is registered in some directory.
- Binding to a services is dynamic. At design time of an application existence of a respective service is not required. It will be discovered and used dynamically.

An example of a JINI service

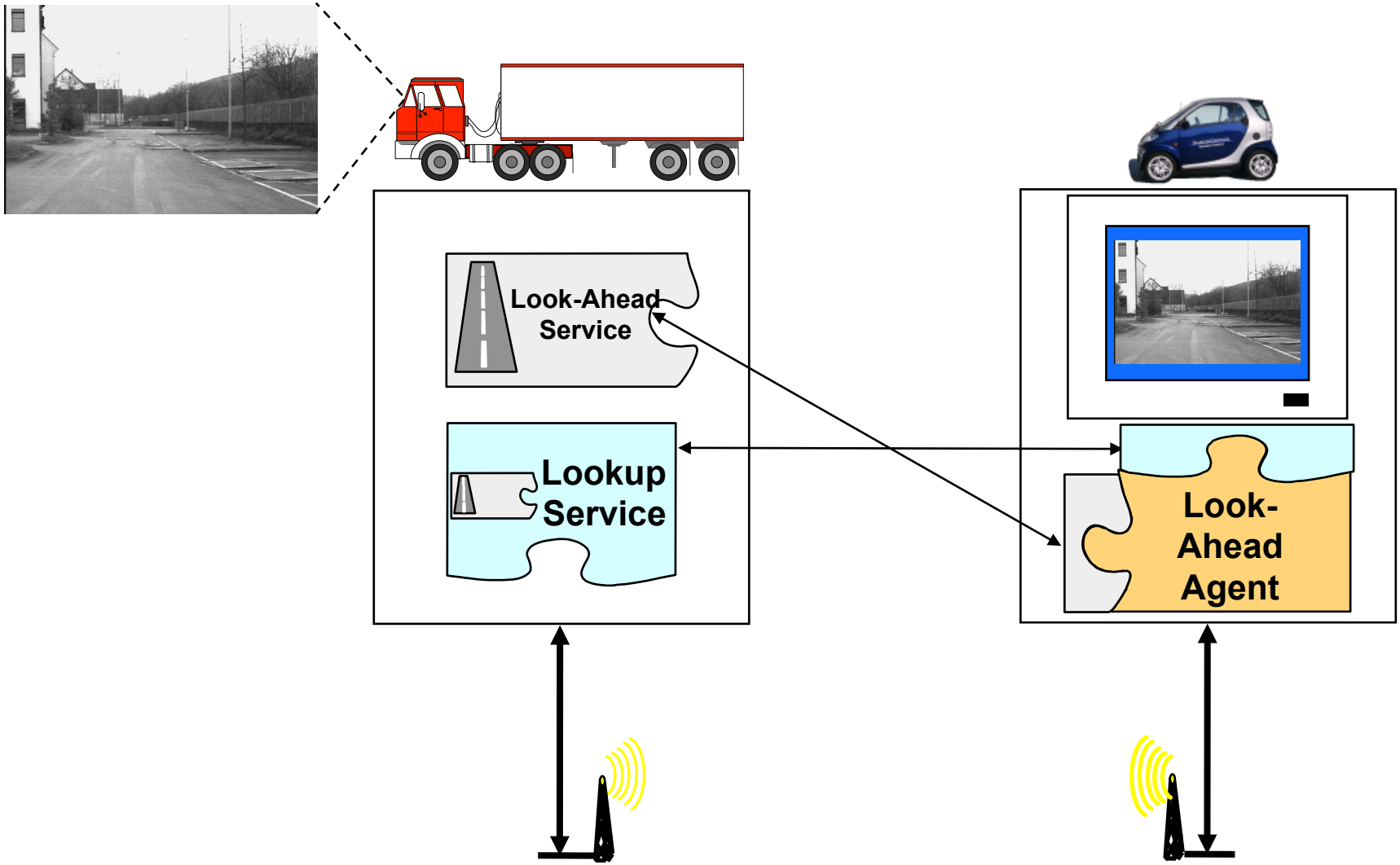
Problems:

1. **How to find a service?**
2. **How to use a service that you have never used before?**
3. **How to deal with server and communication crashes?**

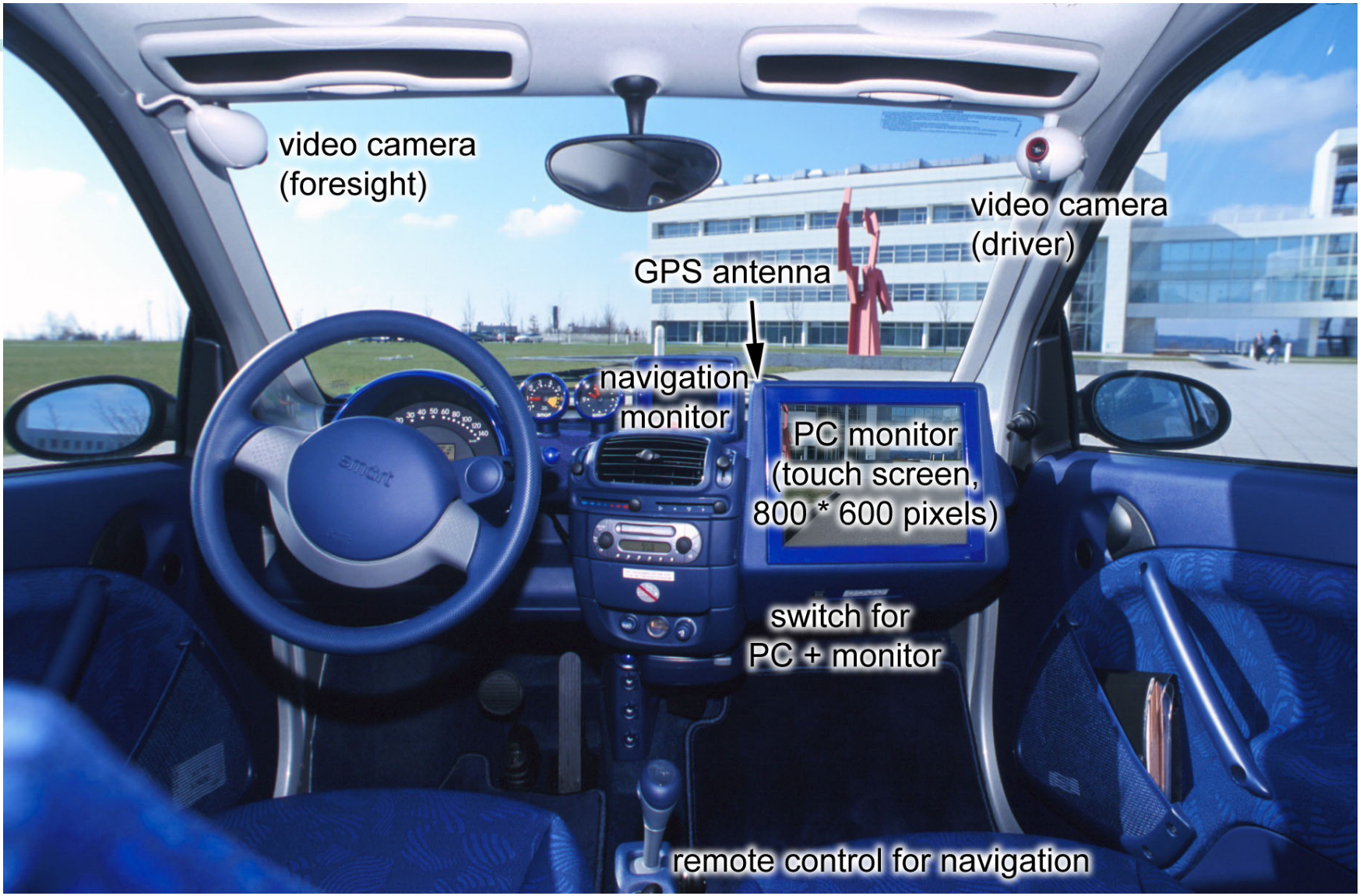
1. Discovery - Finding Lookup Services
2. Join - Service Registration
3. Discovery - Finding Lookup Services
4. Lookup
5. Using a Service

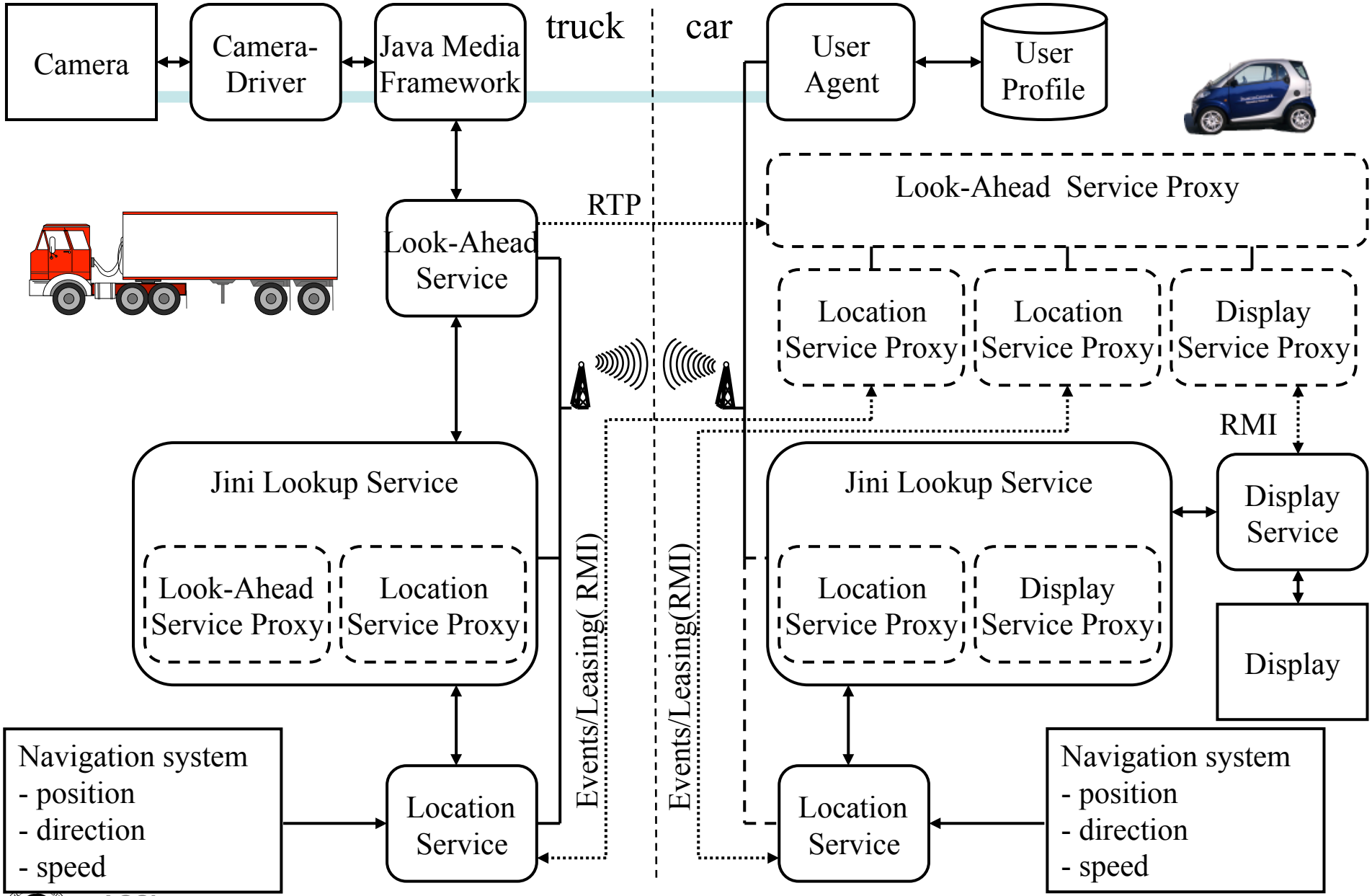


The Demo Scenario: A proactive car-to-car service



The „Hardware“ of the Demo Scenario




Camera
Camera-Driver
Java Media Framework
truck
car
User Agent
User Profile
RTP
Look-Ahead Service
Look-Ahead Service Proxy
Location Service Proxy
Location Service Proxy
Display Service Proxy
RMI
Jini Lookup Service
Look-Ahead Service Proxy
Location Service Proxy
Jini Lookup Service
Location Service Proxy
Display Service Proxy
Display Service
Display
Events/Leasing(RMI)
Events/Leasing(RMI)
Navigation system
- position
- direction
- speed
Location Service
Location Service
Navigation system
- position
- direction
- speed

A bright future

EINE UNFALLURSACHE, DIE AN HÄUFIGKEIT ZUNIMMT:
UNAUFMERKSAMKEIT BEIM FAXEN

Programming models: Actors

Actors and Agents

.. components are concurrent objects that communicate via messaging, rather than abstract data structures that interact via procedure calls. ... We call them *actor-oriented* languages.... Actor-oriented languages, like object-oriented languages, are about modularity of software.

Edward A. Lee, UCB, 2004

The term “actors” was introduced in the 1970’s by Carl Hewitt of MIT to describe autonomous reasoning agents.

The term evolved through the work of Gul Agha and others to refer to a family of concurrent models of computation, irrespective of whether they were being used to realize autonomous reasoning agents.

Actors

General Properties:

- Actors provide a natural generalization for objects
- Actors encapsulate both, code and data.
- Actors differ from sequential objects in that they are units of concurrency.
- Each Actor executes asynchronously and its operation may overlap with other actors
- The unification of data abstraction and concurrency is in contrast to language models where an explicit and independent notion of thread is used to provide concurrency.
- Actors free the programmer from having to write explicit synchronization code to prevent harmful concurrent access to data within an object.

Gul A. Agha, Prasannaa Thati, Reza Ziaei:
Actors, A Model for Reasoning about Open Distributed Systems, 2001

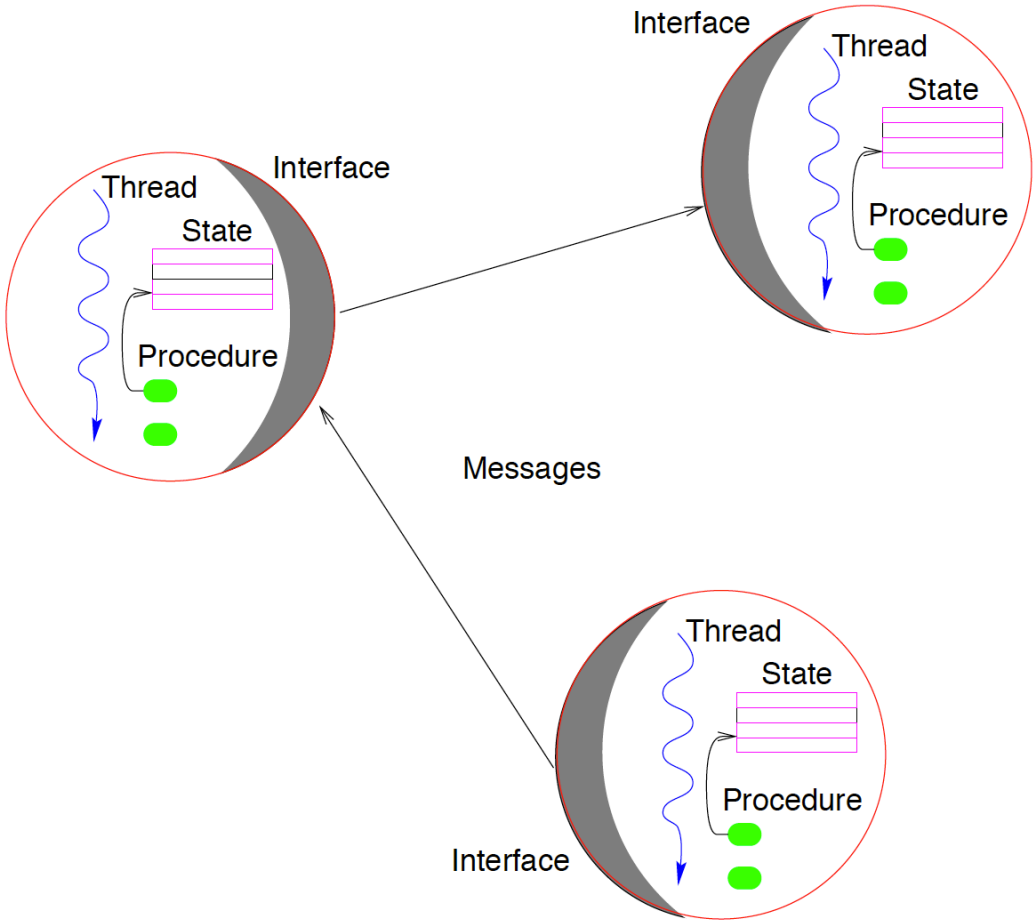
Actors

- Actors have unique and persistent names,
- Actors provide buffered, asynchronous and fair (messages sent are eventually received) communication as a primitive.
- New actors may be created with their own unique and persistent names.

Actor primitives:

- send (a,v): send a messages with content v to actor a;
- create (b): create a new actor with behaviour b;
- ready (b): captures local state change.
  - Selects the behaviour to be used for the next message
  - Prepares the actor to accept another message

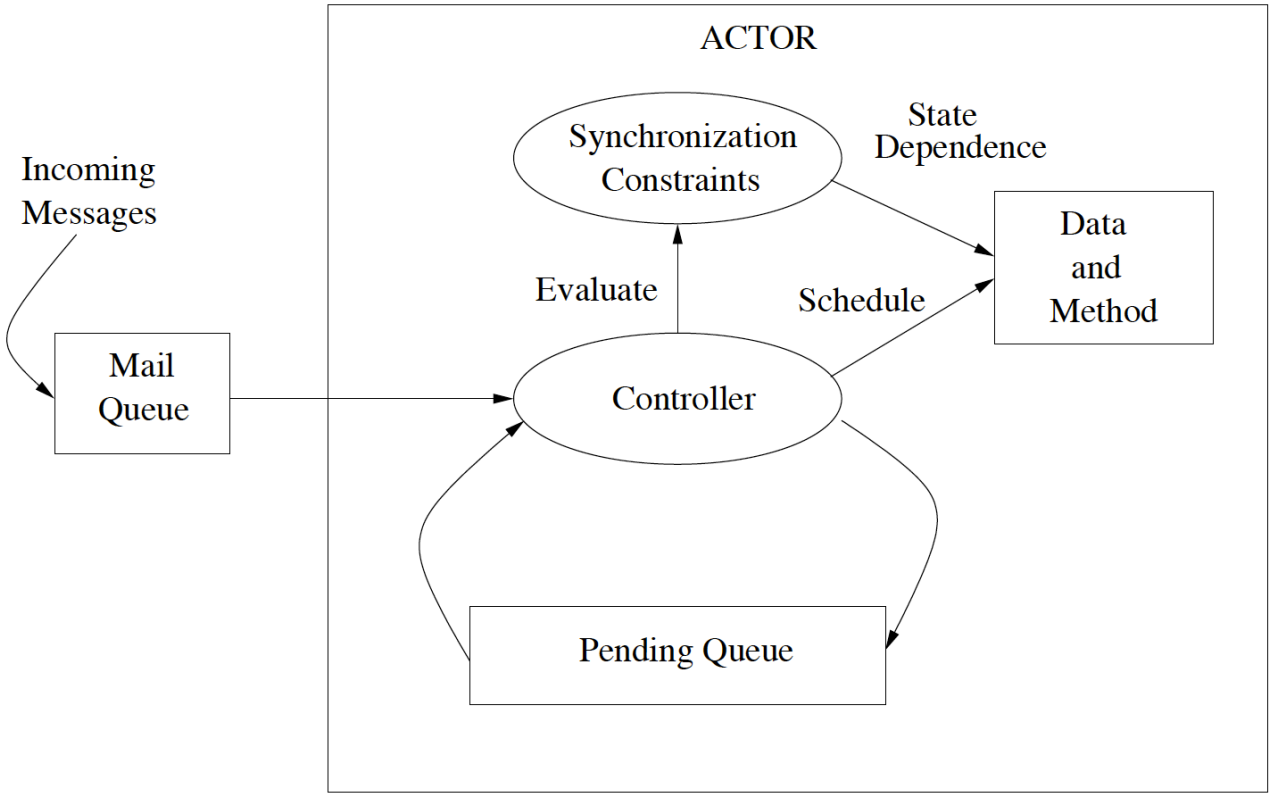
Actors



Actors encapsulate a thread and a state. The interface is comprised of public methods which operate on the state.

Gul A. Agha, Prasannaa Thati, Reza Ziaei:
Actors, A Model for Reasoning about Open Distributed Systems, 2001

Actors



An actor with local synchronization constraints

Gul A. Agha, Prasannaa Thati, Reza Ziaei:
Actors, A Model for Reasoning about Open Distributed Systems, 2001

Actors

Actors are complex, physical, possibly distributed architectural objects that interact with their surroundings through one or more *signal-based* boundary objects called ports.

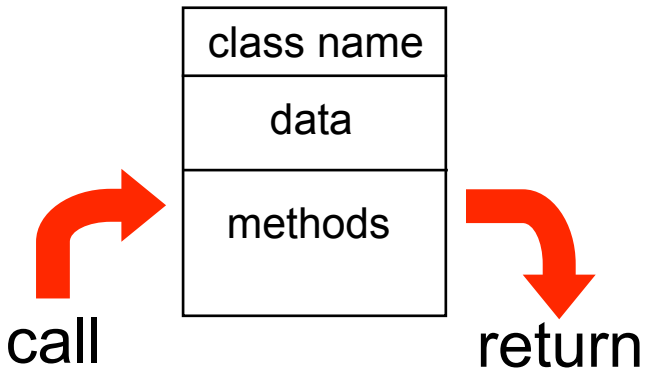
A *port* is a physical part of the implementation of an actor that mediates the interaction of the actor with the outside world. It is an object that implements a specific interface.

Bran Selic, ObjecTime Limited, Jim Rumbaugh, Rational Software Corporation: "Using UML for Modeling Complex Real-Time Systems, March 11, 1998

Objects vs. Actors

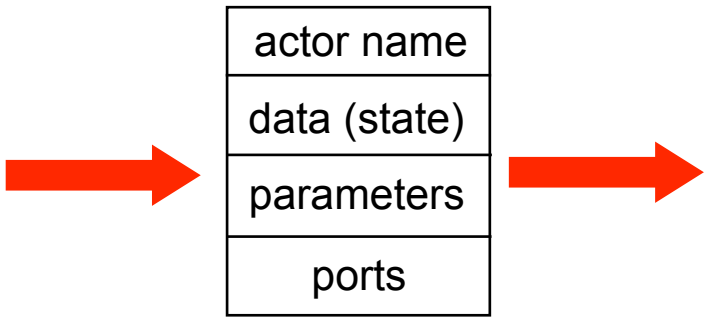
Edward A. Lee, UCB, 2004

Object orientation:



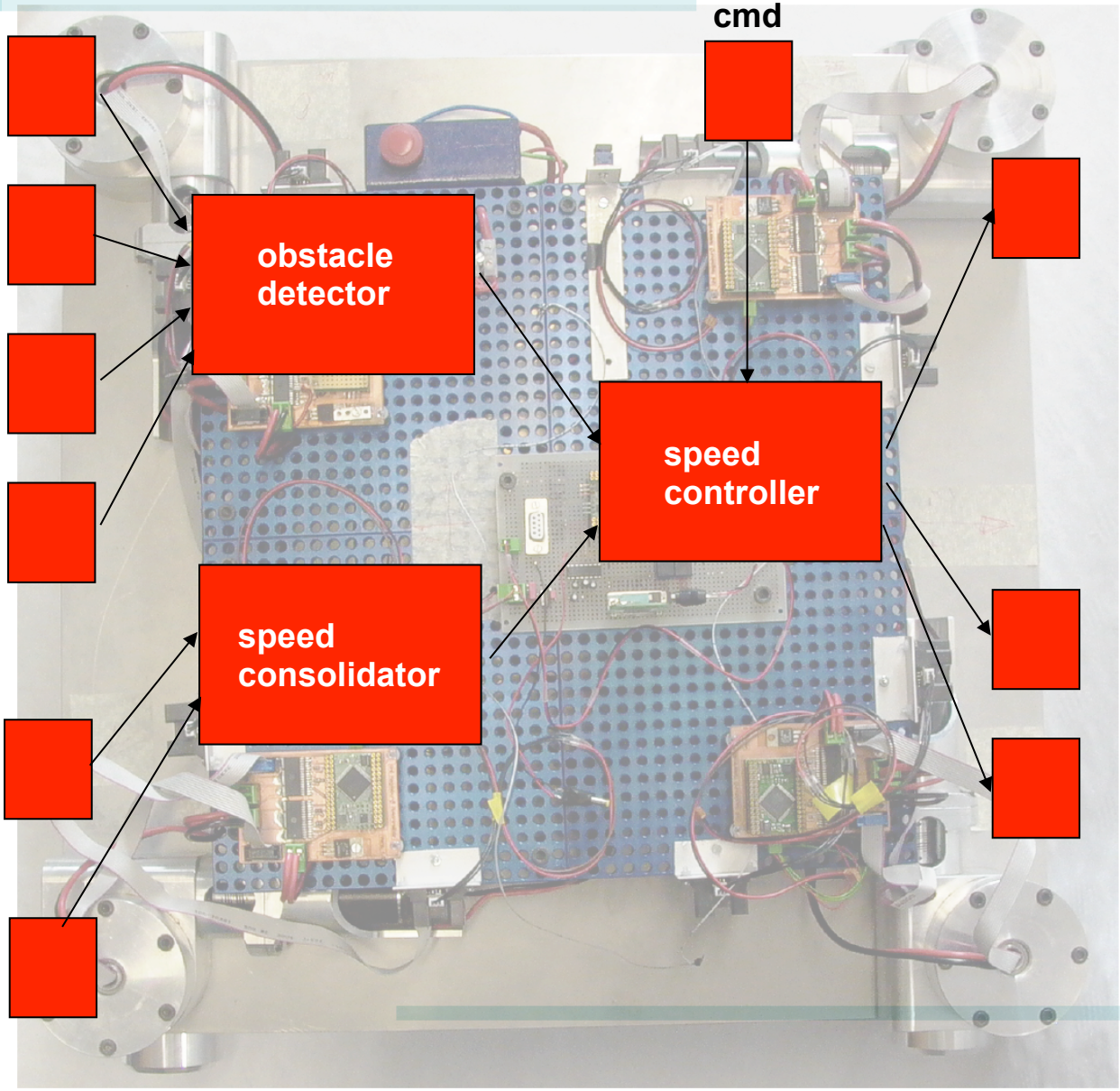
What flows through an object is sequential control

Actor orientation:

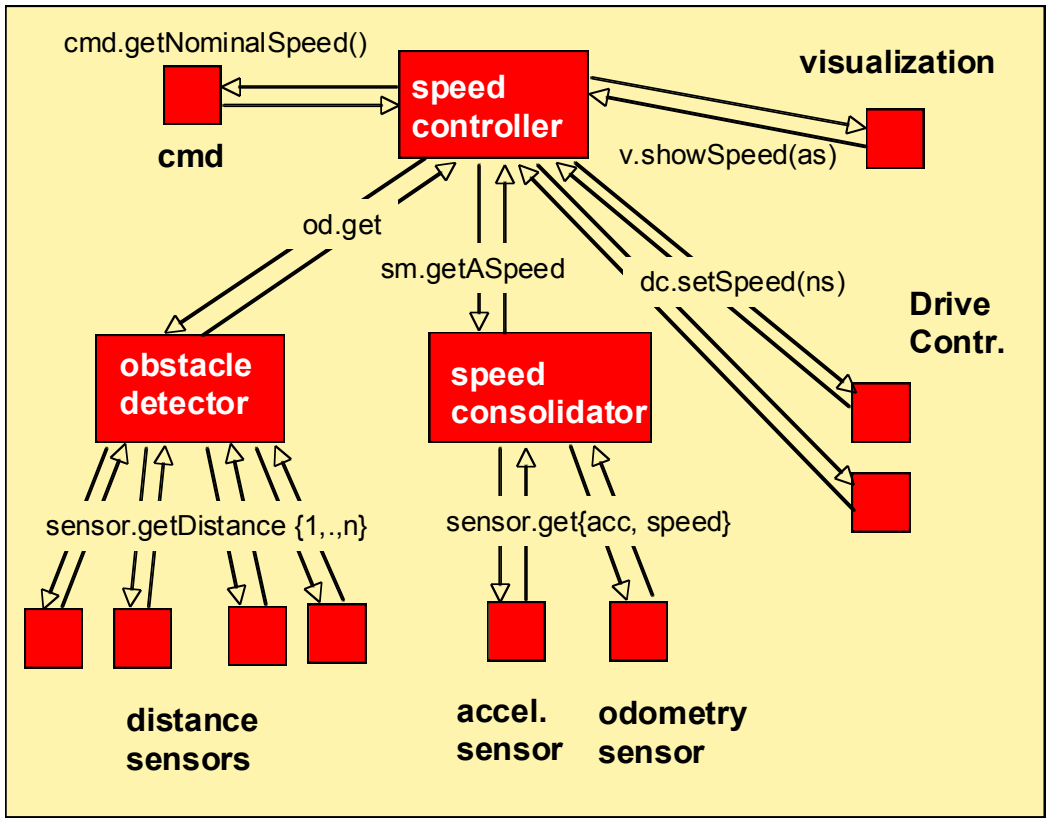


What flows through an actor is streams of data

Sentient Objects A simple control example



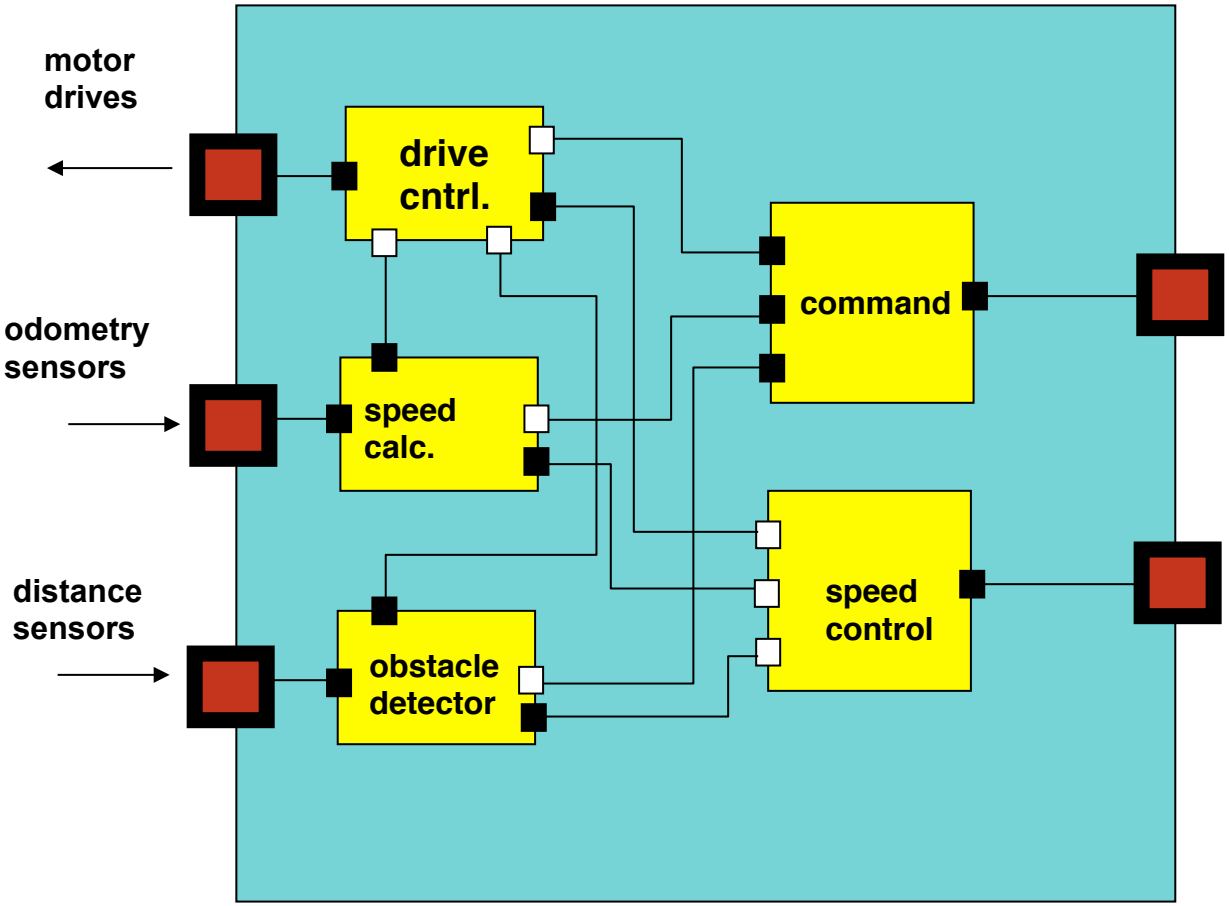
Actors vs. Sequential Objects



C++ code of the sequential control flow:

```
voidSpeedController::control() {
        ns = cmd.getNominalSpeed();
        if (od.get()) {
                ns=0;
        }
        as=sm.getASpeed();
        if (as != ns) {
                dc1.setSpeed(ns);
                dc2.setSpeed(ns);
        }
        v.showSpeed(as);
```

An autonomous component as a “capsule“



Command Port:

- set speed
- set max_speed
- set max_acc
- stop
- go
-
-

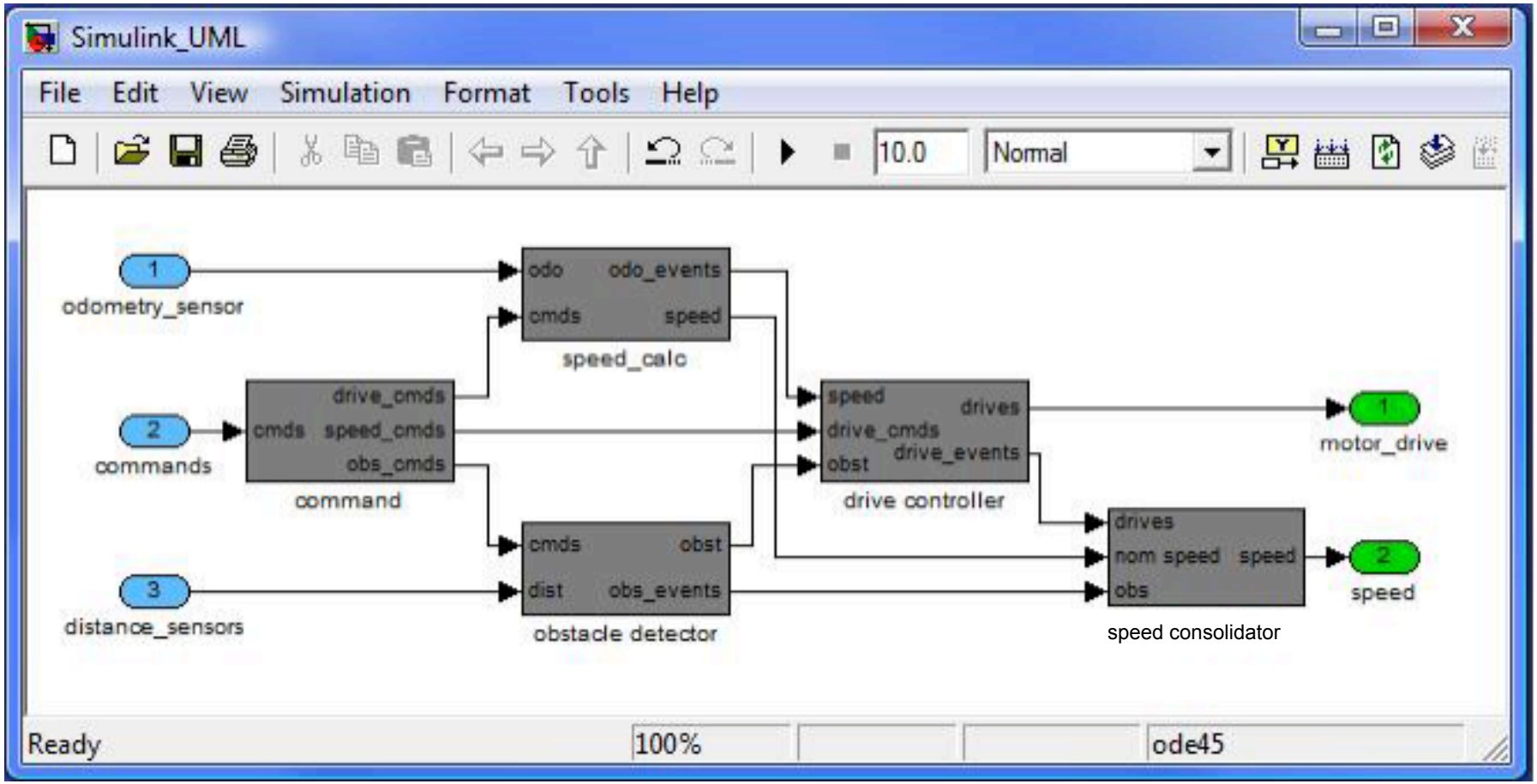
Output Port:

- speed

- max_acc
- low_dist
- alarm_dist
- distance
- acceleration

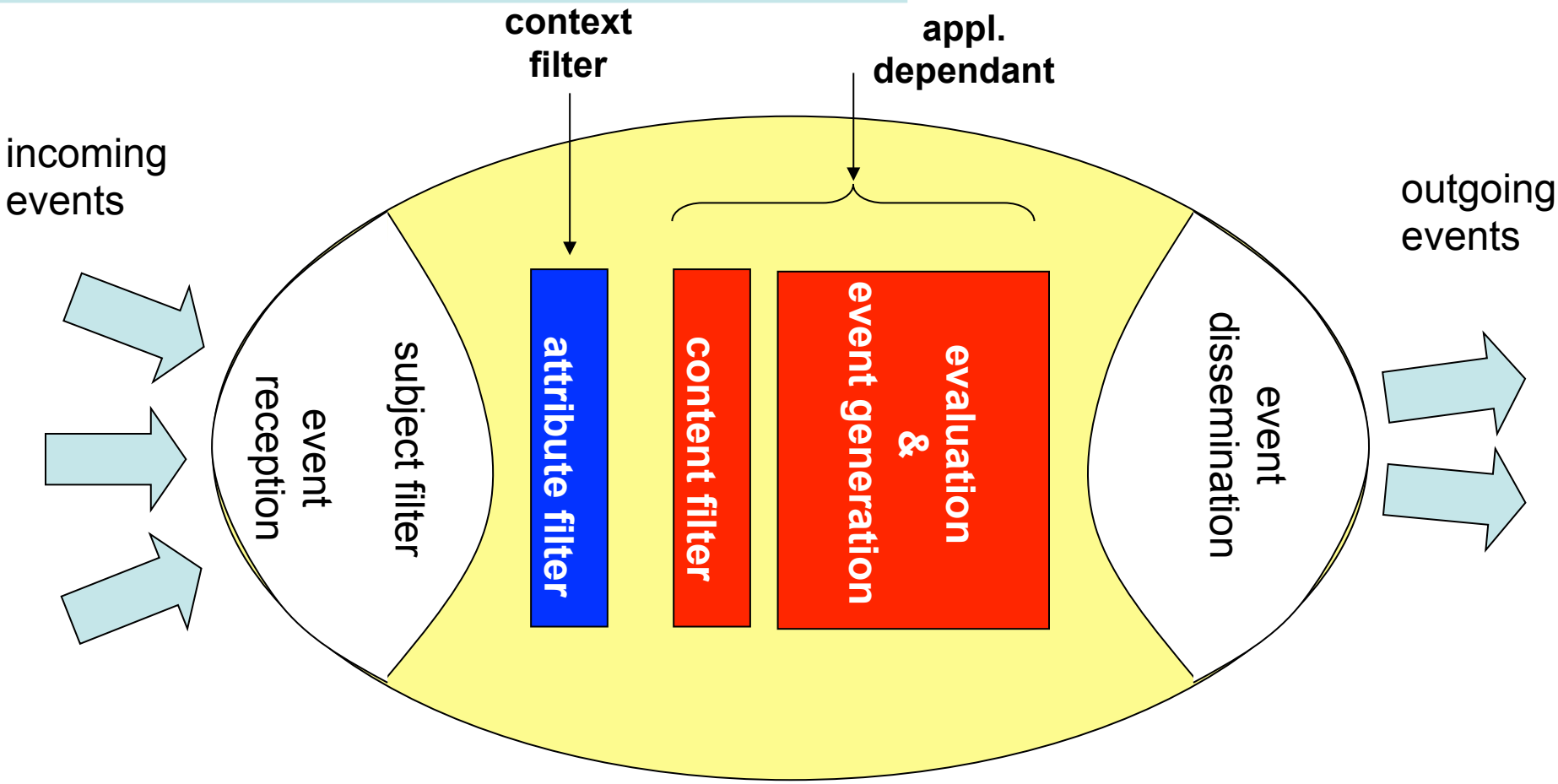
-
-

Actors



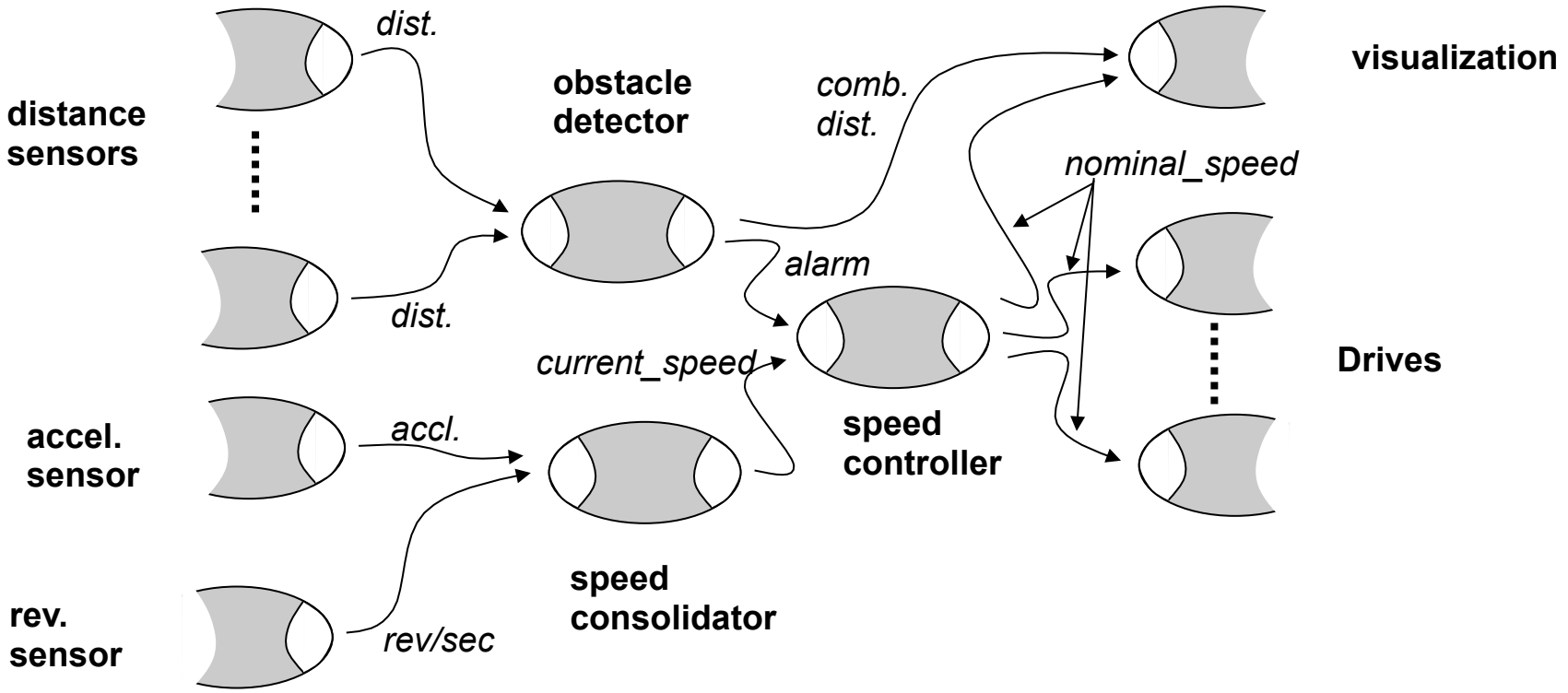
models concurrent flow of information

Sentient Object Structure



Events are typed messages <subject, attr., contents>.
Event channels encapsulate network properties.

A simple control example using COSMIC sentient objects



models concurrent flow of information

Actors vs. Sequential Objects

C++ code of the sequential control flow:

```
voidSpeedController::control() {
        ns = cmd.getNominalSpeed();
        if (od.get()) {
                ns=0;
        }
        as=sm.getASpeed();
        if (as != ns) {
                dc1.setSpeed(ns);
                dc2.setSpeed(ns);
        }
        v.showSpeed(as);
```

C++ code of the modified data flow control program

```
voidSpeedController::controlPeriod() {
        if (as != ns)
                publish(speedCh,ns);
        publish(visSpeedCh,as);    }

voidSpeedController::refresh_as(p_as) {
        as = p_as;                  }

voidSpeedController::refresh_ns(p_ns) {
        if(!od)  ns = p_ns;
else ns = 0;            }
```

Actors and Agents

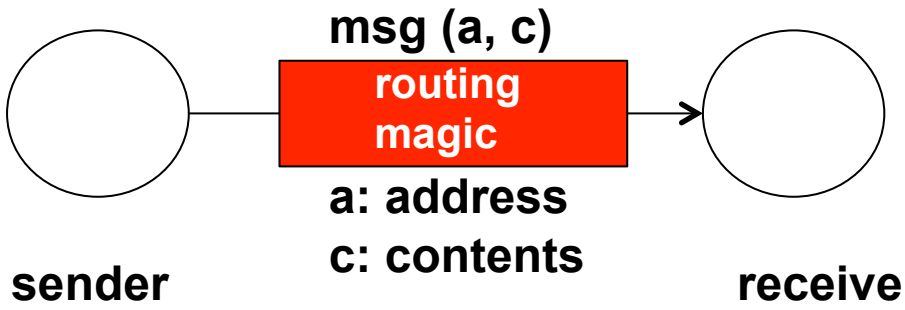
Properies of Agents:

- **Activity: an agent is an actor**
- **Automomy: agents behave according a plan**
- **Social behaviour: ability to communicate (with humans)**
- **Reactivity: an agent reacts on perceived events**
- **Proactivity: agents are able to take initiative**

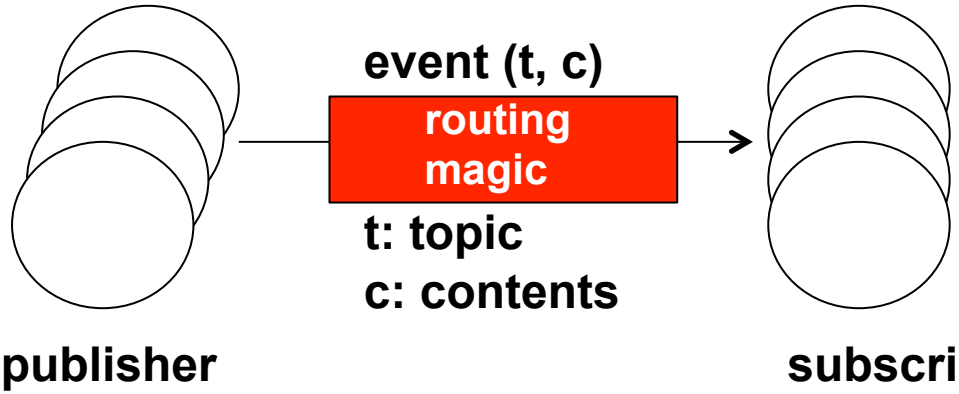
Actors and Agents

"Agents are autonomous, computational entitities that can be viewed as perceiving their environment through sensors and acting upon their environment through effectors. To say that agents are computational entities simply means that they physically exist in the form of programs that run on computing devices. To say that they are autonomous means that to some extent they have control over their behavior and can act without the intervention of humans and other systems. Agents pursue goals or carry out tasks in order to meet their design objectives, and in general these goals and tasks can be suplementary as well as conflicting." (Gerhard Weiß)

Messaging vs. Notification



binding sender-receiver by address at design time



binding sender-receiver by topic or contents at run time

needs more careful examination