Session 3: Distributed Programming

Nodes, Processors, Messages and Simulation Configuration

Tobias Baumgartner

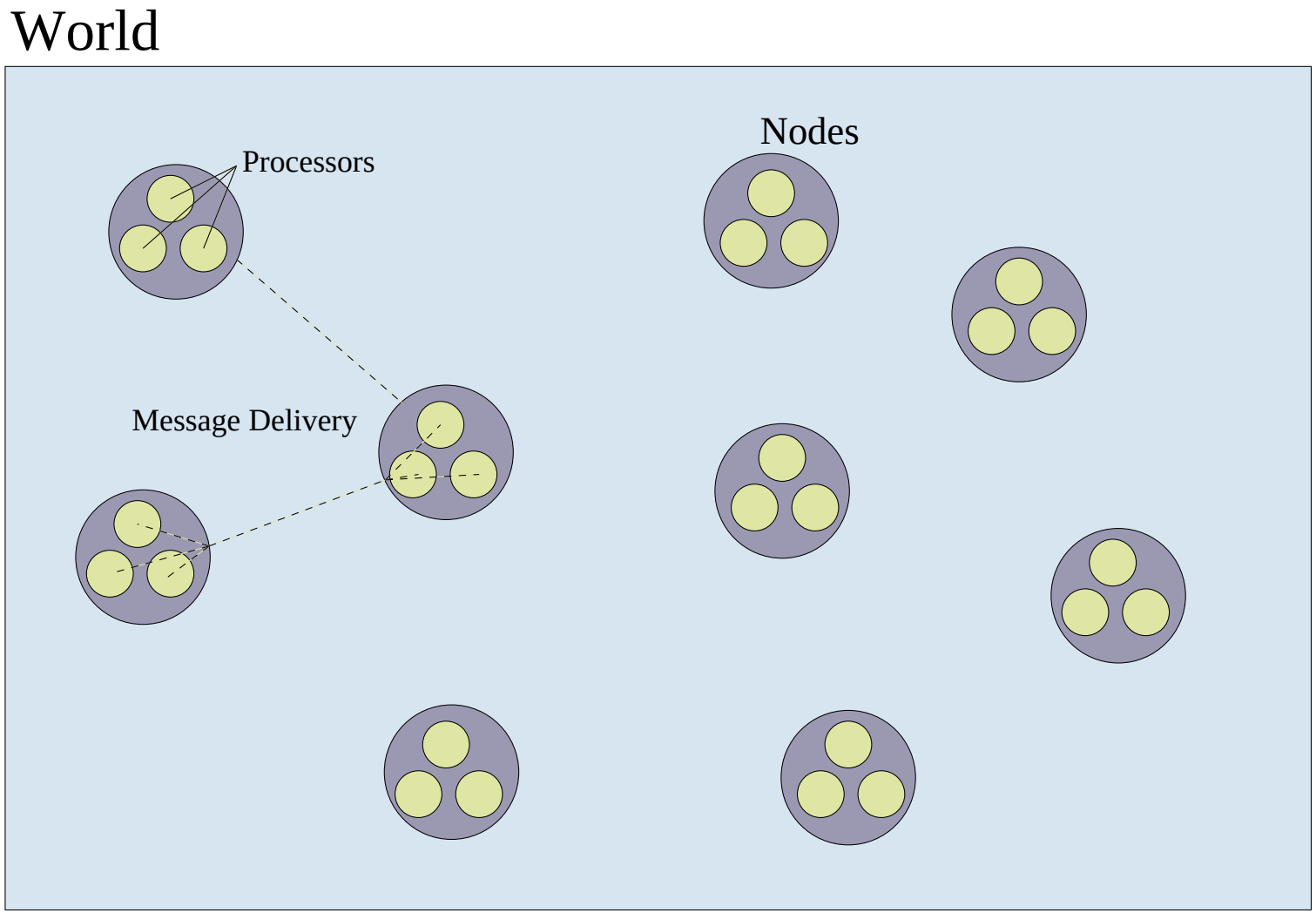
Shawn Winter School 2008, 17.-21.11.08, Lübeck, Germany

November 18th, 2008

Outline

Outline

Core components for Distributed Programming



Outline

World: Basic Functionality

- Container for the simulated sensor network
  - Size of Area
  - Contains all nodes
  - Contains basic Models
- Status of simulation
  - Current time
  - Number of (in)actives nodes
- Provide Access to Nodes
  - Node Iterators
  - Searching certain nodes
  - Neighborhood of nodes

Node Iterator

```
1   for( World::const_node_iterator
2          it = world().begin_nodes();
3          it != world().end_nodes();
4         ++it )
5     const Node& node = *it;
6
7   for( World::node_iterator
8          it = world().begin_nodes_w();
9          it != world().end_nodes_w();
10        ++it )
11    Node& node = *it;
```

Writable and Const

If class members can be accessed via methods, these methods follow a certain naming rule. If the return value is writable, the method gets the suffix $*\_w$. Const return values do not have any suffix.

Outline

Nodes: Basic Functionality

- Container for processors → Each node can contain multiple processors
- Send and Receive messages
- Position in the World

```
1  Vec real_position();
```

- Unique label

```
2  string label();
```

- ID provided, but **not** to be used to store references to nodes. In dynamic networks, IDs may change during simulation.

```
3  int id()
```

- Exactly one node is special node, e.g. the gateway

```
4  bool is_special_node()
```

- Provide access to the World

```
5  const World& world()
6  World& world_w()
```

Outline

Processors: Basic Functionality

- Basis for own implementations of distributed algorithms
- Possible to attach multiple processors to a Node, e.g.
  - Routing Algorithm
  - Time Synchronization
  - ...
- Send and Process messages
- Can adopt three states
  - Active ... *fully functional*
  - Sleep ... *do not receive messages*
  - Inactive ... *can only be reactivated externally*
- If all processors in a node are inactive $\rightarrow$ Node gets inactive
- If all nodes in the world are inactive $\rightarrow$ Simulation finished

The Processor API (1 of 3)

Implemented by developer, if needed

void boot()

Called once for each processor before the simulation starts. Initialization of the application.

void special_boot()

Called once for every processor contained in the special node. This method is called before *boot()*.

void work()

Called once in each simulation round. Used to perform perodic tasks.

The Processor API (2 of 3)

Implemented by developer, if needed

bool process_message(MessageHandle)

Called when a Node receives a message. Handled to each processor, until one returns `true`.

Can receive different message types, so check for right type.

```
1  bool
2  MyProcessor::
3  process_message( const shawn::ConstMessageHandle& mh )
4  {
5      const MyMessage* mymsg =
6          dynamic_cast<const MyMessage*> ( mh.get() );
7      if ( mymsg != 0 )
8      {
9          // handle my message
10         return true;
11     }
12
13     return shawn::Processor::process_message( mh );
14 }
```

The Processor API (3 of 3)

Can be used by developer

void send(MessageHandle)

Enqueues a message for sending.

```
1    send( new MyMessage );
```

void set_state(Processor::State)

Sets the state of the processor

- *Processor::Active*
- *Processor::Sleeping*
- *Processor::Inactive*

const Node& owner()

Node& owner_w()

Returns the *Node* that contains the processor. Provides access to the nodes functionality.

Outline

The Message Class

- Abstract Superclass for all messages
- Every protocol implements its own message type(s)
- On reception type is identified by casting to type
  $\longrightarrow$ No limititation due to global type ids or the like
- Most important methods:

  source()
  : Node that sent the message

  size()
  : Possibility to set the size of the message, e.g. in bytes. Must be set by user.

  timestamp_time()
  : Exact time when message has been sent

Process of Message Sending

1. Processor sends a Message (MyMessage, derived from Message)

```
send( new MyMessage );
```

2. Node sets `source` and `timestamp`, pass to Transmission Model
3. Transmission Model delivers to neighboring nodes
4. On each node, pass to processors
5. On processor, try to cast to own message type.

```
const MyMessage* mymsg =
    dynamic_cast<const MyMessage*> ( messagehandle.get() );
```

If successful, return `true`

Outline

Handles (1 of 2)

- **Problem:**
  When creating an instance with `new`, who calls `delete` and where is it called?
- **Solution:**
  Using the provided Handles that do the deletion after there isn't any handle left to the object.
- **Concept:**
  Pass the newly created object directly to the Handle via

```
1 MyObjectHandle = new MyObject;
```

- **Hint:**
  Get object from handle via `get()`

```
2 MyObject *obj = MyObjectHandle.get();
```

- **Attention:**
  Use Handles consistently! Do not try to hold own pointers of handled objects persistently.

Handles (2 of 2)

- **Example:**
  The previously seen Message is an example for this concept.
  - Processor awaits a Handle in `send`

```
1 void send( const MessageHandle& );
```

  - If someone sends a message, a new instance is passed

```
2 send( new MyMessage );
```

  - Deletion of `MyMessage` is done automatically.
- **Writing your own Handle:**
  There are mainly two instructions to be followed.
  - Class must be derived from `RefcntPointable`
  - Call macro `DECLARE_HANDLES(MyClass)` that declares
    - `MyClassHandle`
    - `ConstMyClassHandle`

Outline

Keeper-Concept

- Goal: Flexibility in accessing different objects at runtime
- Concept: Map with string as key, and object as value
- Such Objects must be `KeeperManaged`, they provide

  string name(void)

  `string` that is used as the key in the map
- Keeper provides

  void add(KeeperManaged value)

  Add a new value that in turn is identified by its `name()`

  KeeperManaged& find_w(string name)

  Return the value that is mapped to the key `name`
- SimulationController provides access to keepers
  - `SimulationTaskKeeper`
  - `ProcessorFactoryKeeper`
  - ...

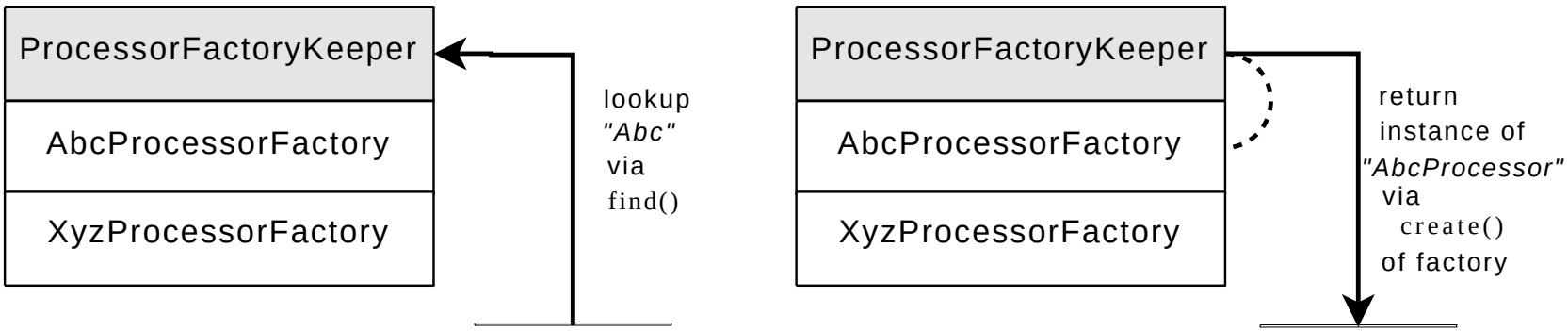
Factory-Classes

- Goal: Scalability in creating objects
- Concept: Implementing factories
  - Responsible for creating objects
  - Common interface, implement method `create()`
  - Factories are generally held in a Keeper
- Used, for example, for generating *n* Processors for *n* Nodes

Processor Factories

Whenever a user develops a processor , a relating factory must be implemented that returns an instance of the processor via `create()`. The factory in turn must be registered at the `ProcessorKeeper` of the simulation controller, usually in init of application.

Example: Processor Factory Keeper



Outline

Configuration via Parameters

- Parameters can be set in configuration, e.g. key=value
- Values can be Integer, String, Boolean and Double
- Stored in the *SimulationEnvironment*

```
1 SimulationEnvironment& se =
2   owner().world().simulation_controller().environment();
```

- Parameters can be read out by two methods

```
3 value = se.required_int_param( "KEY" );
4 value = se.optional_int_param( "KEY", DEFAULT_VALUE );
```

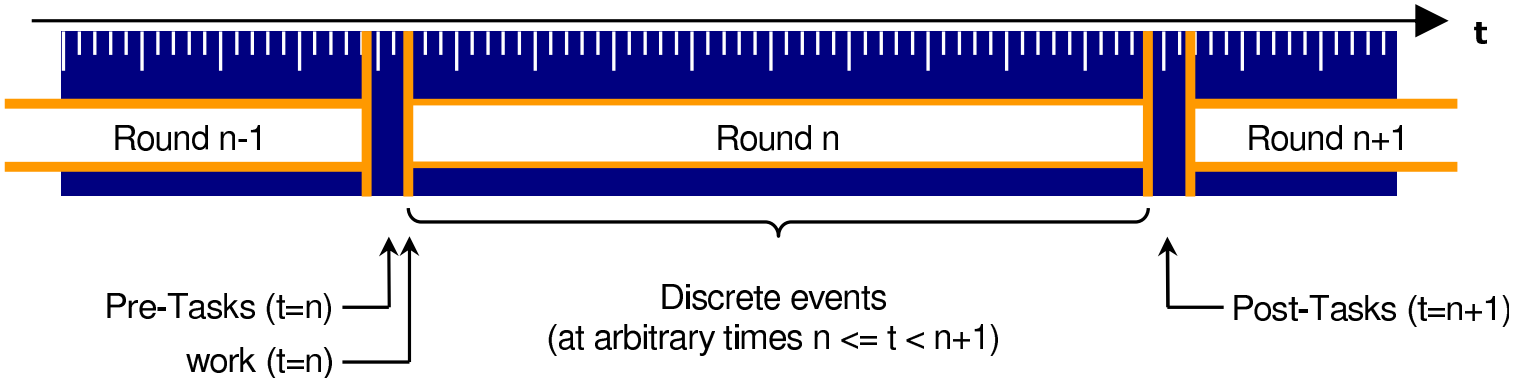
- Replace `int` with `string`, `bool`, or `double`

Note

Configuration parameters can not be read in constructor of a processor, because the *owner*-node is not available. Use `boot()`-method instead.

Outline

Time in the Simulation



- Continuous timeline, with arbitrary small resolution
- Events can be registered at any time with precision of `double`
- For simplifying and performance reasons:
  - Also division in rounds
  - Coarse but efficient models available that rely on rounds
  - Simulation tasks can be registered as *pre-step* or *post-step* tasks
  - At the beginning of each round the nodes' work methods are invoked

The Event Scheduler

- Triggers the execution of events
- Objects derived from *EventHandler* can register at the Event Scheduler to be notified at the given point in time

```
1 EventHandle new_event(
2     EventHandler& eh, double t, const EventTagHandle& eth);
```

- Simulation always skips to the next event time and notifies the registered handlers → Avoid useless steps

How to use the EventScheduler

1. Derive the Simulation object from the class *EventHandler* and implement the main interface method *timeout()*

```
1  class MySimulationObject :
2  public shawn::EventScheduler::EventHandler
3  { ...
4    void timeout (
5      EventScheduler&, EventHandle, double, EventTagHandle& )
6    ... }
```

2. Register the event at the *EventScheduer*

```
1  /* Creates a new event that timeouts at time t
2     Use the EventTagHandle to tag information to the event */
3  EventHandle EventScheduler::new_event(
4     EventHandler& eh, double t, const EventTagHandle& eth )
```

3. The event can be rescheduled or deleted

```
1  //Reschedules the event for a different time
2  void move_event (EventHandle, double)
3  //Removes the specified event
4  void delete_event (EventHandle)
```

Control flow of one Simulation Round

1. Set $round = round + 1$
2. Execution of Pre-Step-Tasks
3. Simple transmission models deliver messages
4. Run `work`-Methods of Nodes
5. Playback of scheduled events for time t, with $round \leq t < round + 1$
6. Execution of Post-Step-Tasks

Outline

Exercises

- Development of a simple, distributed application
- Use the existing ws08 template in legacyapps
- Use `http://wisebed.eu/ws08/ws08-session03.conf`
- Tree Routing Algorithm
  1. Gateway floods the network in the beginning
  2. Each node stores its predecessor wrt Gateway
  3. Each node stores hop-count to Gateway
  4. Each node periodically sends State-Messages to Gateway with current hop-count
  5. Message is routed over predecessors to gateway

Hints

- Use different types of messages, e.g.
  - Flooding message
  - Routing message
- Processor can store its predecessor as

```
1  const shawn::Node* predecessor_;
```

- Use special boot method to identify gateway:

```
1  virtual void special_boot( void ) throw();
```

- For periodic events, use either `work()` method or Event Scheduler