CORBA Components Demonstration

Frank Pilhofer

February 21, 2002

Abstract

The CCM Implementors Group intends to demonstrate availability of several implementations of the CORBA Component Model at the OMG Meeting in Yokohama (April 2002). This document defines the demo scenario as well as the hardware environment. The demo will focus on the interoperability of existing components.

1 Procedures

Frank Pilhofer will act as the coordinator for the demonstration. Participants that want to join the demonstration shall contact the coordinator. Parties that are unable to come to Yokohama but still want to demonstrate their implementation shall register through a participant that agrees to host their implementation. The demonstration will take place on Tuesday afternoon; a preliminary meeting will be held on Monday for a live test.

2 Hardware Environment

The demonstration will be run on multiple computers interconnected by a TCP/IP network. There will be a 100 MBit Ethernet Hub; each participant shall bring their own compouter and a "10 Base T" cable of appropriate length (3 meters or more). Each participant will be assigned one or more static IP address. Participants that wish to hook up more than one computer shall bring additional hubs.

I will bring some extension cords to provide sockets for all of us. Please bring adapters that match german sockets (in other words, I will use a single japanese socket / german plug adapter and german extension cords), or else be prepared to bring some extension cords to directly hook up to Japanese plugs. In any case, the power will be 100 Volts at 50 Hertz.

3 Software Environment

We do not expect that interoperable assembly and deployment tools will be available in time. Therefore, the demonstration will be limited to the interoperability of existing (already deployed) components. All participants will deploy components using their own, vendor-specific tools, and then register their homes in a common Naming Service. Of course, each participant is welcome to demonstrate their deployment software in a separate scenario.

Each participant will be provided with a `corbaloc:` or `http:` style object reference of the Root Naming Context. Each participant will then create a new Naming Context with the name of their implementation as the new node's name. Within that Naming Context, each participant will register all homes required by the demonstration. Each home will be registered using its IDL name in the `name` field; the `kind` field shall be left empty, or shall carry a sequential number if you wish to run multiple homes of the same type.

A tool will be provided that traverses the Naming Service, creating components from their homes, interconnecting them, and then calling `configuration_complete` on each component's equivalent interface.

```
module DiningPhilosophers {
  exception InUse {};

  interface Fork {
    void get () raises (InUse);
    void release ();
  };

  component ForkManager {
    provides Fork the_fork;
  };

  home ForkHome manages ForkManager {};
};
```

Figure 1: IDL for the ForkManager Component

4 Scenario

The "Dining Philosophers" scenario will serve as an example.

A number of Philosophers is sitting around a circular table. Between each two is a fork, so there is an equal number of philosophers and forks. Philosophers think until they become hungry, then they need both forks (the one on their left and the one on their right side) in order to eat. Obviously, this can cause conflicts, because there are not enough forks for all philosophers to eat at the same time. An unlimited amount of food on each plate is assumed. Some means of fairness must be implemented to avoid deadlocks, livelocks and starvation.

Philosophers and forks will be implemented as components; forks are facets offered by a ForkManager component; fork references are then used by the philosopher components.

At regular intervals, philosophers shall publish an event that contains status information. An observer component will consume these events to produce a global status display.

4.1 ForkManager Component

The `ForkManager` component offers a single facet by the name of `the_fork` of type `Fork`. The `Fork` interface has two operations called `get` and `release`. The full IDL for the `ForkManager` component is shown in figure 1.

A hungry philosopher, in acquiring a fork, will call the fork's `get` operation. This will lock the fork for other use until the philosopher calls the fork's `release` operation. If a fork is already in use, and the `get` operation is called again, the operation will throw the `InUse` exception.

Note that this scenario assumes fair philosophers that do not call `release` on a fork that they have not locked themselves.

4.2 Philosopher Component

A `Philosopher` component has two receptacles; one of type `Fork` for each fork (left and right). In addition, a philosopher has an event source port to publish events of type `StatusInfo`. The full IDL for the Philosopher component is shown in figure 2.

Philosophers are autonomous in that they act by themselves rather than just reacting to invocations from the outside. Their behavior can be partially described by a state machine as shown in figure 3. State transitions happen at discrete points in time, called *ticks*. There is at most one state transition per tick.

The "real time" per tick is a per-philosopher constant; each philosopher shall select a random time between 1000 and 3000 milliseconds for each tick.

```
module DiningPhilosophers {
  component Philosopher {
    attribute string name;
    uses Fork left;
    uses Fork right;
    publishes StatusInfo info;
  };

  home PhilosopherHome manages Philosopher {
    factory new (in string name);
  };
};
```

Figure 2: IDL for the Philosopher Component

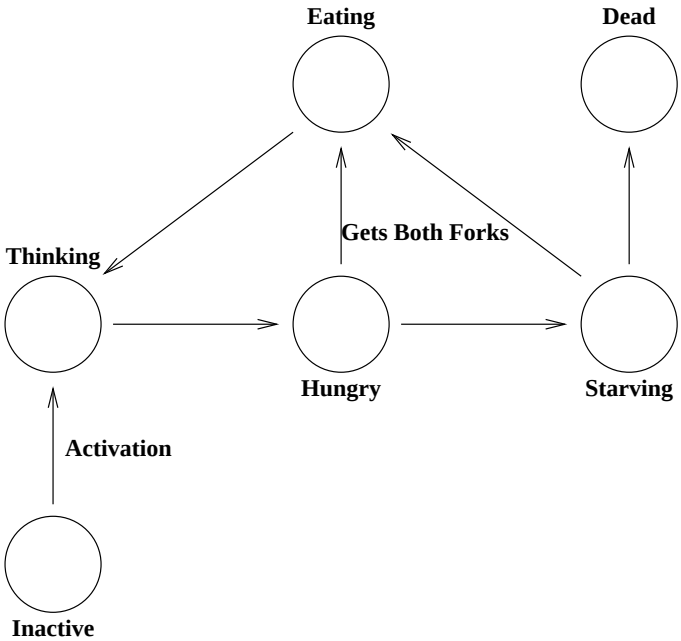

Figure 3: Philosopher State Machine

Note: The rationale behind using a different, random tick interval for different philosophers is that this is an easy measure to avoid livelocks, which might happen if all philosophers pick up and drop forks simultaneously. You could say that philosophers have an individual metabolic rate.

Each tick, philosophers publish a `StatusInfo` event on their `info` port.

In the following, each state is described:

Inactive After deployment and before activation, philosophers are inactive. Upon activation (i.e. after `configuration_complete` has been called on the external interface and `ccm_activate` has been called on the internal interface), the philosopher enters the Thinking state.

Thinking Upon entering the Thinking state, the philosopher initializes an internal counter to zero. This counter is incremented each tick while in the Thinking, Hungry and Starving states and therefore measures the philosopher's "hungryness." This value is later used to compute the time spent eating, and also for possible display in the observer. In the Thinking state, the philosopher does nothing (but publishing `StatusInfo` events). After three (3) ticks, the philosopher enters the Hungy state.

Hungry In the Hungry state, the philosopher would like to eat, so the philosopher attempts to acquire both forks. If the philosopher succeeds getting both forks, the philosopher enters the Eating state. If the philosopher does not succeed getting both forks within seven (7) ticks (i.e. if the hungryness counter becomes 10), the Starving state is entered.

To prevent deadlocks, a philosopher is not allowed to hold a single fork for an extended amount of time; if a philosopher decides holding one fork while waiting for the other, the fork held by the philosopher shall be released after at most five (5) ticks.

Starving In the Starving state, the philosopher acts the same as in the Hungry state, but may implement more desperate measures to hold on to a fork for an extended amount of time while waiting for the other. If the philosopher succeeds getting both forks, the Eating state is entered. If the philosopher does not succeed getting both forks within thirty (30) ticks (i.e. if the hungryness counter becomes 40), the Dead state is entered.

Dead Upon entering the Dead state, the philosopher releases all forks. There is no way out of the Dead state. However, a dead philosopher keeps sending `StatusInfo` events.

Eating In the Eating state, the philosopher holds on to both forks. While in the Eating state, the hungryness counter is decremented by three (3) each tick, so that the philosopher spends one-third of the time eating that the philosopher has been thinking, hungry or starving. If the hungryness counter reaches zero, the philosopher drops both forks and enters the Thinking state.

This section has introduced a number of constants, and the notion of a variable metabolism. The choice of constants has proven effective for the author's implementation of the example (i.e. the philosophers can run for an extended amount of time without dying). Configuring the constants might be intellectually interesting, to see how well the philosophers will perform with a given set of parameters, but that would add little value to our demonstration.

4.3 StatusInfo Event

The `StatusInfo` eventtype carries information that each philosopher publishes each tick. The IDL for the `StatusInfo` eventtype is shown in figure 4.

The fields should be self-explanatory. The `ticks_since_last_meal` field is the same as the hungryness counter that was mentioned in the previous section (i.e. initialized to zero upon entering the Thinking state, then incremented each tick). This value can be used by the observer to display the hungryness in a graphical manner.

```
module DiningPhilosophers {
  enum PhilosopherState {
    EATING, THINKING, HUNGRY, STARVING, DEAD
  };

  eventtype StatusInfo {
    public string name;
    public PhilosopherState state;
    public unsigned long ticks_since_last_meal;
    public boolean has_left_fork;
    public boolean has_right_fork;
  };
};
```

Figure 4: IDL for StatusInfo event

```
module DiningPhilosophers {
  component Observer {
    consumes StatusInfo info;
  };

  home ObserverHome manages Observer {};
};
```

Figure 5: IDL for Observer Component

4.4 Observer

The Observer component has a single event sink port to consume `StatusInfo` events that are published by each philosopher each tick. The IDL for the observer component is shown in figure 5. The observer may display the information in any way it wants.

5 Running the Demo

As mentioned, there will be a tool that finds out about existing components (i.e. their homes) by traversing the single Naming Service, where all homes need to be registered.

The tool will create an equal number of philosophers and forks, e.g. using one philosopher and one fork from each party, depending on the number of participants and the desired number of philosophers.

One instance of the observer component will be created for each participant, which is then subscribed to all philosophers, so that all participants can monitor the philosophers' states simultaneously.

Once running, the demo is expected to run in perpetuity, without manual intervention. However, frequent restarts are likely, if only to demonstrate the deployment and start-up process. Participants should therefore prepare scripts to restart the demo quickly (i.e. starting daemons, deploying components, Naming Service registration).