

Insights in Partial Failures

Leased object references and Unit Testing

Elisa Gonzalez Boix
egonzale@vub.ac.be

Distribution Model = OO + Events



Generate and receive application requests

```
obj<-msg(arg)  
def msg(param) { ... }
```



Follow-up on outstanding requests

```
when: future becomes: { |result| ... }
```



React to services appearing and disappearing

```
when: type discovered: { |ref| ... }
```



React to references disconnecting, reconnecting, expiring

```
when: ref disconnected: { ... }
```

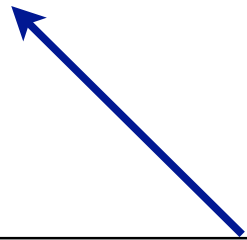
```
when: ref reconnected: { ... }
```

```
when: ref expired: { ... }
```

Conditional Synchronization (CS)

- with Futures:

```
def testAsyncOneCustomer(){  
  def future := when: customer<-haveComputer()@FutureMessage becomes:{  
    |val|  
    self.assertEquals(val, 1);  
  };  
  future;  
};
```

- 
- applying the `becomes:` block resolves future.
 - applying the `catch:` block ruins future.



CS with Futures

- Synchronization based on event or conditions by explicit future manipulation:

```
def [future, resolver] := makeFuture();  
consumer<-give(future);  
def val := /* calculate useful value */  
    resolver.resolve(val);  
    resolver.ruin(exception);
```



CS in Unit Tests

```
def InstantMessengerTest(){
  extend: /.at.unit.test.UnitTest.new("IMTest") with: {
    def test := self;
    def testAsyncMessageSend() {
      def [fut,res] := /.at.lang.futures.makeFuture();
      def IMGUI := object: {
        def init(im) { im.setUsername("Aact") };
        def display(text) {
          test.assertEquals("Added buddy: Bact", text);
          res.resolve(true);
        };
      };
      def A := createIM(IMGUI);
      // ... code to create peer B called Bact
    };
    fut;
  };
};
```

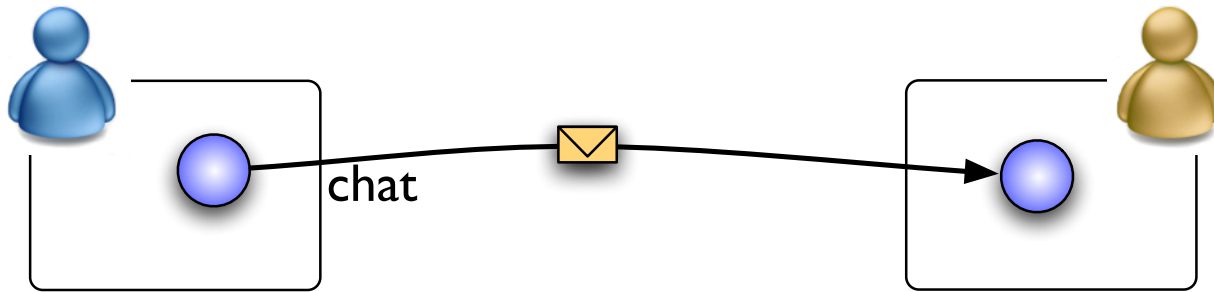


CS in Unit Tests

```
def MyTest(){  
  extend: /.at.unit.test.UnitTest.new("MyTest") with: {  
  
    def testAsyncWhenElapsed() {  
      when: 2.seconds elapsedWithFuture: {  
        self.assertEquals(3,1+2);  
      };  
    };  
  };  
  
>MyTest.runTest()
```



Failure Handling



```
whenever: InstantMessenger discovered: {IchatI
```

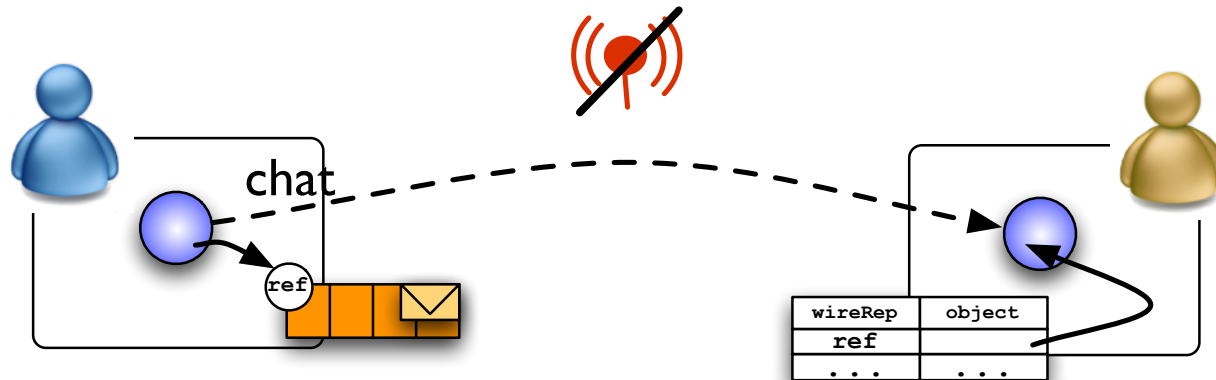
```
...
```

```
when: chat disconnected: {  
  system.println("buddy offline");  
}
```

```
when: chat reconnected: {  
  system.println("buddy online");  
}
```

```
}
```

Far References & Disconnections

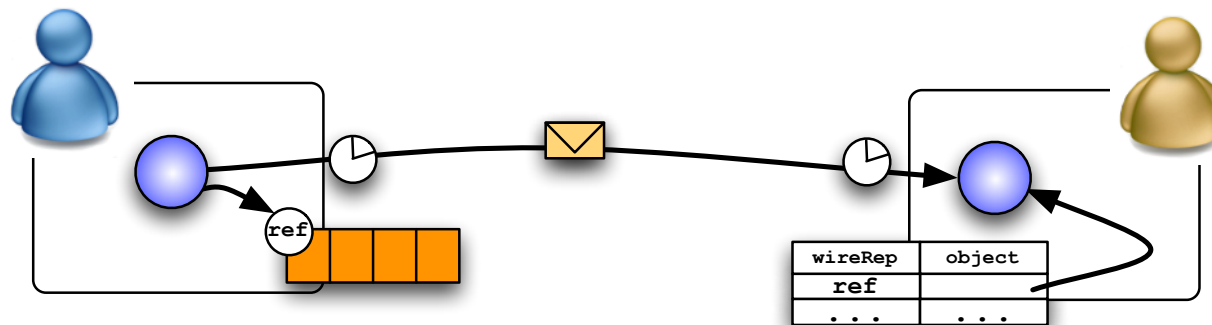


- Far references keep referring the remote object upon a disconnection!

Permanent Disconnections

- But, the system cannot distinguish transient from permanent disconnections.
- Limite lifetime of the remote object references:

Leasing [grey89, waldo01]



Enabling Leased Refs

```
import /.at.lang.leasedrefs;
```

```
lease: timeout for: object
```

- object serialization returns a leased object reference rather than a far reference.

```
def session := lease: minutes(30) for: (  
  object: {  
    def joinSession(player) {...}  
    def receiveVote(poll){...}  
  }  
);
```



Working with Leased Refs

```
import /.at.lang.leasedrefs;
```

- leasing integrated with remote references.

```
def session := object: {  
  def addItemToCart(anItem) { ... }  
  def checkOutCart() { ... }  
};  
def leasedSession := lease: 1.minutes for: session;
```

object serialization returns a leased far reference rather than a far reference.



Working with Leased Refs

- Communication via asynchronous message passing, except for `==` method.
- Managing life cycle of a leased object reference:

renew: leasedRef **for:** timeInterval

revoke: leasedRef

leaseTimeLeft: leasedRef



Working with Leased Refs

- Registering a closure that is executed when the leased reference expires:

```
when: session expired: {  
    system.println("session with " + remotePeer + timed out.");  
    //cleanup code for session  
};
```

- Observers can be placed in both client and server side!
- An expired leased object reference behaves as a permanently disconnected far reference!

Leasing Variants

renewOnCallLease: timeout **for:** object

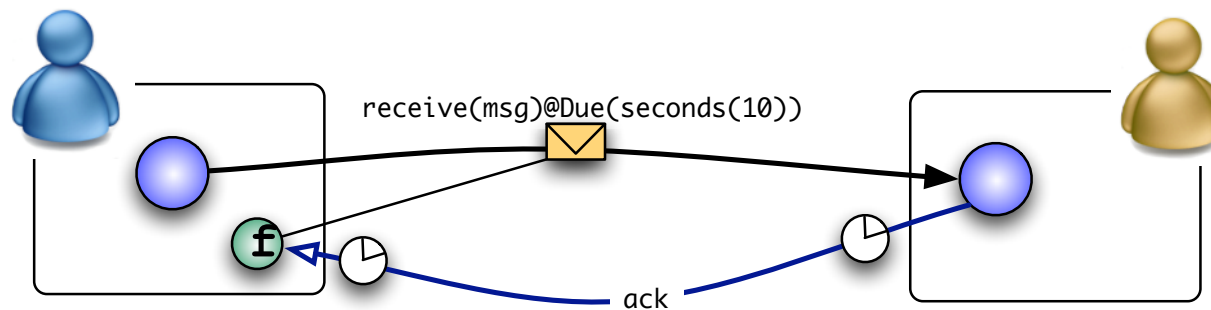
- the leased reference gets renewed as long as it receives messages.

singleCallLease: timeout **for:** object

- the leased reference gets revoked upon a successful method call on the server object.



Leasing and Futures



```
when: buddy<-receive(msg)@Due(seconds(10)) becomes: { |ack|  
  system.println( msg.content + " sent to: " + to)  
} catch: TimeoutException using: { |e|  
  system.println("msg: " + msg.content + " timed out.");  
};
```

- Future is passed as a `singleCallLease` which expires:
 - upon reception of a `resolve` or `ruin` message
 - due to a timeout