Beernet: RMI-free peer-to-peer networks

Boris Mejías, Alfredo Cádiz, Peter Van Roy
Département d'ingénierie informatique
Université catholique de Louvain, Belgium
{firstname.lastname}@uclouvain.be

ABSTRACT

The key issue in distributed programming is partial failure: how to handle failures of part of the system. This unavoidable property causes uncertainty because we cannot know whether a remote object is ever going to reply to a message. It is also the reason why RMI/RPC is difficult to use. In this paper we describe the most convenient object-oriented mechanism we have found to develop peer-to-peer applications effectively, namely by using active objects that communicate via asynchronous message passing and fault streams for failure handling. We show that this works better than the usual approach of using RMI to communicate and distributed exceptions for failure handling. We define our peers as lightweight actors and we use them to build a highly dynamic peer-to-peer network that deals well with partial failure and non-transitive connectivity. We give many code examples to show the simplicity and naturalness of our approach.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Languages, Design

Keywords

actors, message-passing, distributed-programming

1. INTRODUCTION

The goal of distributed computing is to achieve the collaboration of a set of different processes. A process is an abstraction of an entity that can perform computations. This entity can be a computer, a processor in a computer, or a thread of execution in a processor. In order to achieve the collaboration of processes, there are several programming paradigms aiming to help developers to build distributed systems. One definition of a distributed system is given by Tanenbaum and van Steen [13]:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system"

This definition suggests using *distribution transparency*, where all the effort of distributed programming is moved to the construction of a middleware that supports the distribution of the programming language entities. But network and computer failures cause unexpected errors to appear at higher abstraction levels, which breaks transparency and complicates programming.

There are four main concerns that make distributed programming harder than a sequential program running in a single process. They are clearly described by Waldo et al. [16], and they involve *latency*, *memory access*, *concurrency* and *partial failure*. Latency is not a critical problem because it does not change the semantics of performing an operation on a local or a distributed entity. It just makes things go a bit slower. Memory access is solved by using a virtual machine that abstracts the access, and then it does not change the operational semantics either. A more difficult problem is concurrency. The middleware has to guarantee exclusive access to the state in order to avoid race conditions. There are different techniques such as data-flow, monitors or locks, that makes possible the synchronization between processes achieving a coherent state. Given that, and even though it is not trivial to write concurrent programs correctly, it is not a critical problem either. What really breaks transparency is partial failure. Basically, distribution transparency works as long as there is no failure.

A partial failure occurs when one component of the distributed system fails and the others continue working. The failure can involve a process or a link connecting processes, and the detection of such a failure is a very difficult task. In distributed environments such as the Internet, it is impossible to build a perfect failure detector because when a process p stops responding, another process p' cannot distinguish if the problem is caused by a failure on the link connecting process p or the crash of the process p itself. This explanation might be trivial, but it is usually forgotten. Failures are a reality on distributed systems. Another definition of a distributed system is given by Leslie Lamport:

> "A distributed system is one in which the fail-

DO21 '09 Genova, Italy

ure of a computer you did not even know it existed can render your own computer unsuable"

Even though this definition does not describe the possibilities of a distributed system, it makes explicit why distributed computing is special.

The classical view of distributed computing sees partial failure as an error. For instance, a remote method invocation (RMI) on a failed object raises an *exception*. This approach actually goes against distribution transparency, because the programmer is not supposed to make the distinction between a local and a distributed entity. Therefore, an exception due to a distribution failure is completely unexpected, breaking transparency. Another less fundamental issue but still relevant, is that RMI and RPC are conceived as synchronous communication between distributed processes. Due to network latency, synchronous communication is not able to provide good performance because the execution of the program is suspended until the answer (or an exception) arrives.

New trends in distributed computing, such as ambient intelligence and peer-to-peer networks, see partial failure as an *inherent characteristic* of the system. A disconnection of a process from the system is considered normal behaviour, where the disconnection could be a gentle leave, a crash of the process, or a failure on the link. We believe that this approach leads to more realistic language abstractions to build distributed systems.

In this paper we discuss the design decisions we have made to build our peer-to-peer network Beernet [11]. Our approach is based on asynchronous message passing to communicate between processes, and on actors [1] as components to organize every process, providing encapsulated state and avoiding shared-state concurrency. We discard the use of RPC or RMI between class-based objects, and we avoid raising exceptions due to broken distributed entities. We have made these decisions based on the needs of building Beernet. We believe that a peer-to-peer network is an interesting case study for distributed systems, because it is highly dynamic with respect to connectivity between peers and because it does not have a central point of control. Both these properties make the management of the system more complex.

Our model is influenced by the programming languages Oz [10] and Erlang [3], and by the algorithms of the book "Introduction to Reliable Distributed Programming" [7], which presents an event-driven model that we describe in the following section. Section 3 presents incrementally the language abstractions we have used in our approach. Section 4 summarizes the related work discussed along the paper. Section 5 finishes by recapitulating our main conclusions.

2. EVENT-DRIVEN COMPONENTS

The algorithms for reliable distributed programming presented in [7] are designed in terms of components that communicate through events. Every component has its own state, which is encapsulated, and every event is handled in a mutually exclusive way. The model avoids shared-state concurrency because the state of a component is modified by only one event at the time.

Every component provides a specific functionality such as point-to-point communication, failure detection, best effort broadcast, and so forth. Components are organized in layers where the level of the abstraction is organized bottom-up. A higher-level abstraction requests a functionality from a more basic component by triggering an event (sending a request). Once the request is resolved, an indication event is sent back to the abstraction (sending back a reply). Algorithm 1 is taken from the book, where only the syntax has been slightly modified. It implements a best-effort broadcast using a more basic component, (*pp2p*), which provides a perfect point-to-point link to communicate with other processes.

Algorithm 1 Best Effort Broadcast

```
1: upon event ⟨ bebBroadcast | m ⟩ do
2:     for all p in other_peers do
3:         trigger ⟨ pp2pSend | p, m ⟩
4:     end
5: end

6: upon event ⟨ pp2pDeliver | p, m ⟩ do
7:     trigger ⟨ bebDeliver | p, m ⟩
8: end
```

The best-effort broadcast (*beb*) component handles two events: *bebBroadcast* as requested from the upper layer, and *pp2pDeliver* as an indication coming from the lower layer. Every time a component requests *beb* to broadcast a message m, *beb* traverses its list of other peers, triggering the *pp2pSend* event to send the message m to every peer p. Every p is a remote reference, but it is the *pp2p* component which takes care of the distributed communication. At every receiving peer, the *pp2p* component triggers *pp2pDeliver* upon the reception of a message. When *beb* handles this event, it triggers *bebDeliver* to the upper layer, as seen in Algorithm 1. It is important to mention that *beb* does not have to wait for *pp2p* every time it triggers *pp2pSend*, and that *pp2p* does not wait for *beb* or any other component when it triggers *pp2pDeliver*. This asynchronous communication between components means that each component can potentially run in its own independent thread.

Using layers of components allows programmers to deal with issues concerning distribution only at the lowest layers. For instance, the component *beb* is conceived only with the goal of providing a broadcast primitive. The problem of communicating with a remote processes through a point to point communication channel is solved in *pp2p*. If a process p crashes while the message is being sent, it does not affect the code of *beb*, thus improving the transparency of the component. There is no need to use something like

try $\langle send\ m\ to\ p \rangle$ **catch** $\langle failure \rangle$

It is the responsibility of *pp2p* to deal with the failure of p. It is also possible that *pp2p* triggers the detection of the crash of p to the higher level, and then it is up to *beb* to do something with it, for instance, removing p from the list of other peers to contact. In such a case, the failure of p is considered as part of the normal behaviour of the system, and not as an exception. Even though the code for the maintenance of *other_peers* set is not given [7], we can deduce it from the implementation of the other components. In Algorithm 2 the *register* event is a request made from the upper layer, and *crash* is an indication coming from the *pp2p* layer.

Even though we advocate defining algorithms using event-driven components using the approach of [7], there are some important drawbacks to consider. To compose layers, it is necessary to create a channel, connect the components using the channel, and subscribe them to the events they will handle. We find this approach a bit over sophisticated. It

Algorithm 2 Best Effort Broadcast extended

```
1: upon event ⟨ bebRegister | p ⟩ do
2:     other_peers := other_peers ∪ {p}
3: end

4: upon event ⟨ crash | p ⟩ do
5:     other_peers := other_peers \ {p}
6: end
```

could be simplified by talking directly to a component and using a default listener only when necessary. A related problem concerns the naming convention of events. The name reflects the component implementing the behaviour, making the code less *composable*. For instance, if we want to use a fair-loss point-to-point link (*flp2p*) instead of *pp2p*, we would have to change the *beb* code by replacing *pp2pSend* by *flp2pSend*, and instead of handling *pp2pDeliver* we would have to handle *flp2pDeliver*.

Since the architecture considers components and channels, an alternative and equivalent approach would be to use objects with explicit triggering of events as method invocation, instead of using anonymous channels. Using objects as collaborators, they could be replaced without problems as long as they implement the same interface. In such an approach, both *flp2p* and *pp2p* would handle the event *send* and trigger *deliver*.

The other problem of [7] is that there is no explanation of how to transfer a message from one process to the other. The more basic component *flp2p* is only specified in terms of the properties it holds, but it is not implemented. There is no language abstraction to send a message to a remote entity.

3. BEERNET

Beernet [11] is a library implemented in Mozart/Oz [10] that provides an API to build peer-to-peer applications. Mozart is an implementation of the Oz language, which is a multi-paradigm programming language supporting functional, concurrent, object-oriented, logic and constraint programming paradigms [15], and offering support for distributed programming with a high degree of transparency. Thanks to the multi-paradigm support of Oz, we were able use more convenient language abstractions for distribution and local computing while building Beernet. In this section we discuss the basic language abstractions that we considered appropriate and necessary to implement event-driven components, and which abstractions allowed us to improve the approach towards an event-driven actor model.

The peer-to-peer network built by Beernet uses the relaxed-ring network topology [8]. It provides a distributed hash table (DHT) with replicated storage using distributed transactions to guarantee data consistency. A peer-to-peer network is a very interesting case study of a distributed system because it is very dynamic. Peers are constantly joining and leaving the network, either as graceful leaves or due to failures. It does not use a central point of control and it can be run without relying on an existing routing infrastructure because it provides its own structured overlay network for message routing. Because of the context, during the rest of the paper we use the term peer as equivalent to the previously used process.

3.1 Threads and data-flow variables

One of the strengths of the Oz language is its concurrency model which is easily extended to distribution. The kernel language is based on procedural statements and single-assignment variables. When a variable is declared, it has no value yet, and when it is bound to a value, it cannot change the value. Attempting to perform an operation that needs the value of such a variable will wait if the variable has no value yet. In a single-threaded program, that situation will block forever. In a multi-threaded program, such a variable is very useful to synchronize threads. We call it a data-flow variable. Oz provides lightweight threads running inside one operating system process with a fair thread scheduler.

The code in algorithm 3 shows a very simple example of data-flow synchronization. First, we declare variables *Foo* and *Bar* in the main thread of execution. Then, a new thread is created to bind variable *Bar* depending on the value of *Foo*. Since the value of *Foo* is unknown, the '+' operation waits. A second thread is created which binds variable *Foo* to an integer. At this point, the first thread can continue its execution because the value of *Foo* is known.

Algorithm 3 Threads and data-flow synchronization

```
1: declare Foo Bar
2: thread Bar = Foo + 1 end
3: thread Foo = 42 end
```

This synchronization mechanism does not need any lock, monitor, or semaphore, because there is no explicit state, and therefore, no risk for race conditions. The values of *Foo* and *Bar* will be the same for all possible execution orders of the threads. Single-assignment variables are also used in languages such as E [9] and AmbientTalk [6, 14], where they are called promises or futures. They are combined with the *when* operator as one of the mechanisms for synchronization.

The execution of a concurrent program working only with single-assignment variables is completely deterministic. While this is an advantage for correctness (race conditions are impossible), it is too restrictive for general-purpose distributed programming. For instance, it is impossible to implement a server talking to two different clients. To overcome this limitation, Oz introduces Ports, which are described in the following section.

3.2 Ports and asynchronous send

A port is a language entity that receives messages and serializes them into an output stream. After creating a port, one variable is bound to the identity of the port. That variable is used to send asynchronous messages to the port. A second variable is bound to the stream of the port, and it is used to read the messages sent to the port. The stream is just like a list in Lisp or Scheme, a concatenation of a head with a tail, where the tail is another list. The list terminates in an unbound single-assignment variable. Whenever a message is sent to the port, this variable is bound to a dotted pair containing the message and a fresh variable.

Algorithm 4 combines ports with threads. First we declare variables P and S. Then, variable P is bound to a port having S as its receiving stream. A thread is created with a *for-loop* that traverses the whole stream S. If there is no value on the stream, the *for-loop* simply waits. As soon as a message arrives on the stream, it is shown on the output console. A second thread is created to traverse a list

of beers (*BeerList*, declared somewhere else), and to send every beer as a message to port P. This is a like a barman communicating with a client. Everybody who knows P can send a message to it, as in the third thread, where the list of sandwiches is being traversed and sent to the same port. Beers will appear on the stream in the same order they are sent. Beers and sandwiches will be merged in the stream of the port depending on the order of arrival, so the order is not deterministic between them.

Algorithm 4 Port and asynchronous message passing

```
 1: declare P S
 2: P = {NewPort S}
 3: thread
 4:     for Msg in S do {Show Msg} end
 5: end
 6: thread
 7:     for Beer in BeerList do {Send P Beer} end
 8: end
 9: thread
10:     for Sdwch in SandwichList do {Send P Sdwch} end
11: end
```

The send operation is completely asynchronous. It does not have to wait until the message appears on the stream in order to continue with the next instruction. The actual message send could therefore take an arbitrary finite time, making it suitable for distributed communication where latency is an issue. With the introduction of ports, it is already possible to build a multi-agent system running in a single process where every agent runs on its own lightweight thread. The non-determinism introduced with ports allows us to work with explicit state, and there is no restriction on the communication between agents.

3.3 Going distributed

Event though full distribution transparency is impossible to achieve because of partial failures, there is some degree of transparency that is feasible and useful. Ports and asynchronous message passing as they are described in the previous section can be used transparently in a distributed system. The semantics of `{Send P Msg}` is exactly the same if P is a port in the same process or in a remote peer. In both cases the operation returns immediately without waiting until the message is handled by the port. If there is a need for synchronization, the message can contain an unbound variable as a future. Then, the sending peer waits for the variable to get a value, which happens when the receiving peer binds the variable. This implies that the variable, and whatever is contained in the message, is transparently sent to the other peer. Variable binding must therefore be transparent.

Algorithm 5 does a ping-pong between two different peers. Code lines from 1 to 5 represent peer A who sends a ping message to peer B. The message contains an unbound variable *Ack*, which is bound by peer B to the value `pong`. Binding variable *Ack* resumes the *Wait* operator at peer A. Peer B, from lines 6 to 10, makes a pattern matching of every received message with pattern `ping(A)`. If that is the case, it binds A to `pong` and continues with the next message. The pattern matching is useful to implement a method dispatcher as we will see in the next section.

This sort of transparency is not difficult to achieve, except

Algorithm 5 Ping-Pong

```
 1: % at Peer A
 2: declare Ack
 3: {Send PeerB ping(Ack)}
 4: {Wait Ack}
 5: {Show "message received"}

 6: % at Peer B
 7: for Msg in Stream do
 8:     case Msg of ping(A) then
 9:         A = pong
10:     end
11: end
```

when a partial failure occurs. An older release of Mozart, version 1.3.0, takes the classical approach to deal with partial failures: it raises an exception whenever an operation is attempted on a broken distributed reference. Most programming languages take the same approach. This approach has two important disadvantages. First, it is cumbersome because it is necessary to add **try** ... **catch** instructions whenever an operation is attempted on a remote entity. More fundamentally, exceptions break transparency when reusing code meant for local ports. If a *distribution* exception is raised, it will not be caught because the code was not expecting that sort of exception.

AmbientTalk [6, 14] adopts a better approach. In ambient-oriented programming, failures due to temporary disconnections are a very common thing, therefore, no exception is raised if a message is sent to a disconnected remote reference. The message is kept until the connection is restored and the message is resent. Otherwise if the connection cannot be fixed after a certain time, it will be garbage collected. Failures are also a common thing in peer-to-peer networks. The normal behaviour of a peer is to leave the network after some time. Therefore, a partial failure should not be considered as an exceptional situation.

A more recent Mozart release, version 1.4.0, does not raise exceptions when distributed references are broken. It simply suspends the operation until the connection is reestablished or the entity is killed. If the operation needs the value of the entity, for instance in a binding, the thread blocks its execution. If a send operation is performed on a broken port, because of its asynchrony, it still returns immediately, but the actual sending of the message is suspended until the connection is reestablished. This failure handling model [5] is based on a *fault stream* that is attached to every distributed entity. An entity can be in three states, *ok*, *tempFail*, or *permFail*. Once it reaches the permanent failure state, it cannot come back to *ok*, so the entity can be killed. If the entity is in temporary failure for too long, it can be explicitly killed by the application and forced to *permFail*. To monitor an entity's fault stream, the idea is to do it in a different thread that does not block and that can take actions over the thread blocking on a failed entity.

3.4 Event-driven Actors

The actor model [1] provides a nice way of organizing concurrent programming, benefiting from encapsulation and polymorphism in analogous fashion to object-oriented programming. We extend the previous language abstractions with Oz *cells* which are containers for mutable state. State

is modified with operator ':=', and it can be read with operator '@'. We do not need to add new language abstractions in order to build our event-driven actors. Without language support, actors are a programming pattern in Oz as is shown in Algorithm 6. Having ports, the cell is not strictly necessary but we use it to facilitate state manipulation. Every actor runs in its own lightweight thread and communicates asynchronously with other actors through ports. Encapsulation of state is achieved with lexical scoping, and exclusive access to state to avoid race conditions is guaranteed by handling only one event/message at a time.

Algorithm 6 is a working implementation of Algorithms 1 and 2 using the language abstractions we have described in this section. It is written in Oz without syntactic support for actors but the semantics are equivalent. The function *NewBestEffortBroadcast* creates a closure containing the state of the actor and its behaviour. The state includes a list of *OtherPeers* and another actor implementing perfect point-to-point communication, which is named *ComLayer* to make explicit that it could be replaced by any actor that understands event *send*, and not only *pp2p*.

The behaviour is implemented as a set of procedures where the signature of the event is specified in each procedure's argument. For instance, the declaration on code line 9 reads that procedure *Receive* implements the behaviour to handle **upon event** *deliver(Src Msg)*. The variable *Listener* represents the actor in the upper layer.

Algorithm 6 Beernet Best Effort Broadcast

```
 1: fun {NewBestEffortBroadcast Listener}
 2:     OtherPeers ComLayer
 3:     SelfPort SelfStream
 4:     proc {Broadcast broadcast(Msg)}
 5:         for Peer in OtherPeers do
 6:             {Send ComLayer send(Peer Msg)}
 7:         end
 8:     end
 9:     proc {Receive deliver(Src Msg)}
10:         {Send Listener Msg}
11:     end
12:     proc {Add register(Peer)}
13:         OtherPeers := Peer | @OtherPeers
14:     end
15:     proc {Crash crash(Peer)}
16:         OtherPeers := {Remove Peer @OtherPeers}
17:     end
18: in
19:     OtherPeers = {NewCell nil}
20:     ComLayer = {NewPP2Point SelfPort}
21:     SelfPort = {NewPort SelfStream}
22:     thread
23:         for M in SelfStream do
24:             case M.label
25:             of broadcast then {Broadcast M}
26:             [ ] deliver then {Receive M}
27:             [ ] register then {Add M}
28:             [ ] crash then {Crash M}
29:             end
30:         end
31:     end
32:     SelfPort
33: end
```

Variable *SelfPort* is bound to the port that will receive all messages coming from other actors. A thread is launched to traverse the *SelfStream*. For every message that arrives on the stream, pattern matching checks the label of the message in order to invoke the corresponding procedure. This part of the code represents the method dispatching of the actor. In the Beernet implementation, the creation of the port and the method dispatching are modularized to avoid code duplication, thus reducing the code size of every actor.

The book [7] contains complementary material including a Java implementation of the *beb* component. Discarding comments and import lines, the implementation takes 67 lines of code, with the component infrastructure already abstracted. It is worth mentioning that a large number of lines are dedicated to catch exceptions. Equivalent functionality within the Beernet actor model takes only 33 lines.

3.5 Peer-to-peer

The architecture of Beernet is based on layers that abstract the different concepts involved in the construction of the peer-to-peer network. A closely related work is the Kompics component framework [2], which follows the component-channel approach of [7] using a similar architecture. The main difference with Beernet is that instead of having components that communicate through channels, we decided to use event-driven actors.

Beernet is built on top of the relaxed-ring [8], a structured overlay network providing a distributed hash table (DHT) as in Chord [12]. In such a network peers are organized into a ring. Hash keys goes from 0 to $N-1$ forming a circular address space. Every peer joins the network with an identifier. The identifier is used to find the correct predecessor and successor in the ring. When peer q joins in between peers p and s, it means that $p < q < s$ following the ring clockwise. Peer s accepts q as predecessor because it has a better key than p. Another reason to be a better predecessor, is that the current predecessor is detected to have crashed. Hence, the maintenance of the ring involves *join* and *crash* events, and it must be handled locally by every peer in a decentralized way.

In order to keep the ring up to date, Chord performs a periodic stabilization that consists in verifying each successor's predecessor. From the viewpoint of the peer performing the stabilization, if the predecessor of my successor has an identifier between my successor and myself, it means that it is a better successor for me and my successor pointer must be updated. Then, I notify my successor. Algorithm 7 is taken from Chord [12]. Only the syntax is adapted. The big problem with this algorithm is seen in line 2. Asking for successor's predecessor is done using RMI. This means that the whole execution of the component waits until the RMI is resolved. There is no conflict resolution if *successor* is dead or dies while the RMI is taking place. If there is a partial failure, the algorithm is broken.

An improved version of the stabilization protocol is given in Algorithm 8 using event-driven actors. The representation of a peer is a data structure having *Peer.id* as the integer identifying the peer, and *Peer.port* as the remote reference, being actually an Oz port. The '.' is not an operator over an actor or an object. It is just an access to a local data structure. The '...' in the algorithm hide the state declaration and the method dispatcher loop. The '<' operator defines the order in the circular address space. We use it here for

Algorithm 7 Chord's periodic stabilization

```
1: upon event ⟨ stabilize | ⟩ do
2:     x := successor.predecessor
3:     if x ∈ (self, successor) then
4:         successor := x
5:     end
6:     successor.notify(self)
7: end
8: upon event ⟨ notify | src ⟩ do
9:     if predecessor is nil or src ∈ (predecessor, self) then
10:        predecessor := src
11:    end
12: end
```

simplicity without changing the semantics of the algorithm.

Stabilization starts by sending a message to the successor with an unbound variable X to examine its predecessor. The peer then launches a thread to wait for the variable to have a value, and once the binding is resolved, it sends a message to itself to verify the value of the predecessor. This pattern is equivalent to the *when* abstraction in E [9] and AmbientTalk [14]. By launching the thread, the peer can continue handling other events without having to wait for the answer of the remote peer. If the remote peer crashes, the *Wait* will simply block forever without affecting the rest of the computation. When the *Wait* continues, the peer sends a message to itself in order to serialize the access to the state with the handling of other messages. Otherwise there would be a race condition.

Algorithm 8 Chord's improved periodic stabilization

```
1: fun {NewChordPeer Listener}
2:     ...
3:     proc {Stab stabilize}
4:         X in
5:         {Send Succ.port getPredecessor(X)}
6:         thread
7:             {Wait X}
8:             {Send Self.port verifySucc(X)}
9:         end
10:    end
11:    proc {Verify verifySucc(X)}
12:        if Self.id < X.id < Succ.id then
13:            Succ := X
14:        end
15:        {Send Succ.port notify(Self))}
16:    end
17:    proc {GetPred getPredecessor(X)}
18:        X = Pred
19:    end
20:    proc {Notify notify(Src)}
21:        if Pred == nil
22:        orelse Pred.id < Src.id < Self.id then
23:            Pred := Src
24:        end
25:    end
26:    ...
27: end
```

Beernet uses a different strategy for ring maintenance. Instead of running a periodic stabilization, it uses a strategy called *correction-on-change*. Peers react immediately when they suspect another peer to have failed. The failed peer is removed from the routing table, and if it happens to be the successor, the peer must contact the next peer in order to fix the ring. To contact the next successor, every peer manages a successor list, which is constantly updated every time a new peer join or if there is a failure.

Algorithm 9 presents part of a *PBeer* actor, which is a Beernet peer. Failure recovery works as follows: when peer P fails, a low-level actor running a failure detector triggers the $crash(P)$ event to the upper layer, where *PBeer* handles it. *PBeer* adds the crashed peer to the crashed set and removes it from its successor list. If the crashed peer is the current successor, then the first node from the successor list is chosen as the new successor. A *notify* message is sent to the new successor. When a node is notified by its new predecessor, it behaves as a Chord node, but in addition, it replies with the *updSL* message containing its successor list. In this way, the successor list is constantly being maintained.

Algorithm 9 Beernet's failure recovery

```
1: fun {NewPBeer Listener}
2:     ...
3:     proc {Crash crash(Peer)}
4:         Crashed := Peer | @Crashed
5:         SuccList := {Remove Peer @SuccList}
6:         if P == @Succ then
7:             Succ := {GetFirst SuccList}
8:             {Send Succ.port notify(Self)}
9:         end
10:    end
11:    proc {Notify notify(Src)}
12:        if {Member Pred @Crashed}
13:        orelse Pred.id < Src.id < Self.id then
14:            Pred := Src
15:        end
16:        {Send Src.port updSL(Self @SuccList)}
17:    end
18:    ...
19: end
```

3.6 Fault streams for failure handling

As described at the end of subsection 3.3, we use a *fault stream* associated to every distributed entity in order to handle failures. An operation performed on a broken entity does not raise any exception, but it blocks until the failure is fixed or the thread is garbage collected. This blocking behaviour is compatible with asynchronous communication with remote entities. In the fault stream model, presented by Collet et al [5, 4], the idea is that the status of a remote entity is monitored in a different thread. The monitoring thread can take decisions about the broken entity, in order to terminate the blocking thread. For instance, there are language abstractions to kill a broken entity so it can be garbage collected.

Algorithm 10 describes how we use the fault stream in the implementation of Beernet. There is an actor in charge of monitoring distributed entities called *FailureDetector*. Upon event *monitor(Peer)*, the actor uses the system operation *GetFaultStream* in order to get access to the status of the remote peer. The fault stream is updated automatically by the Mozart system, which sends heartbeat messages to the remote entity in order to determine its state. When the state

changes, the new state appears on the fault stream. If the connection is working, the state is set to *ok*. If the remote entity does not acknowledge a heartbeat, it is suspected of having failed, and therefore, the state is set to *tempFail*. Since Internet failure detectors cannot be strongly accurate, the state can switch between *tempFail* and *ok* indefinitely. As soon as the state is set to *permFail*, however, the entity cannot recover from that state.

If the state is *tempFail* or *permFail*, the actor triggers the event *crash(Peer)* to the *Listener*, which represents the upper layer. If the state switches back to *ok*, the event *alive(Peer)* is triggered. It is up to the upper layer to decide what to do with the peer. In the case of Beernet, this is described in algorithm 9.

Algorithm 10 Fault stream for failure detection

```
 1: fun {FailureDetector Listener}
 2:    ...
 3:    proc {Monitor monitor(Peer)}
 4:       FaultStream = {GetFaultStream Peer}
 5:    in
 6:       for State in FaultStream do
 7:          case State
 8:          of tempFail then {Send Listener crash(Peer)}
 9:          [] permFail then {Send Listener crash(Peer)}
10:          [] ok then {Send Listener alive(Peer)}
11:          end
12:       end
13:    end
14:    ...
15: end
```

4. DISCUSSION AND RELATED WORK

One of the principles we respect in this paper is to *avoid shared-state concurrency*. We achieve this by encapsulating state, by doing asynchronous communication between threads and processes, by using single-assignment variables for data-flow synchronization, and by serializing event handling with a stream (queue) providing exclusive access to the state. The language primitives of lightweight threads and ports are also present in Erlang [3], and they are not specific to object-oriented programming. Single-assignment variables also appear in E [9] and AmbientTalk [14] in the form of promises, and they are meant for synchronization of remote processes instead of lightweight threads.

The actor model presented here through programming patterns is further developed and supported by E and AmbientTalk. There is one important difference related to the use of lightweight threads. Since they are not supported by these two languages, there is basically only one actor running per process. The actor collaborates with a set of passive objects within the same process. Communication with local objects is done with synchronous method invocation. Communication with other actors, and therefore with remote references, is done with asynchronous message passing. This distinction reduces transparency for the programmer because it establishes two types of objects: local and distributed.

In Beernet, we organize the system in terms of actors only, making no distinction in the send operation between a local and a remote port. Transparency is respected by not raising an exception when a remote reference is broken. There is only one kind of entity, an actor, and only one send operation.

As mentioned in the previous section, Kompics [2] is closely related because it is also a component framework conceived for the implementation of peer-to-peer networks. Instead of using actors for composition, it uses event-driven components which communicate through channels, analogous to events in [7].

5. CONCLUSIONS

We have presented examples in this paper to highlight the importance of partial failure in distributed programming. The fact that failures cannot be avoided has a direct impact on the goal of transparent distribution which cannot be fully achieved. Therefore, it has also an impact on remote method invocation, the most common language abstraction to work with distributed objects. Because of partial failure, it is very difficult to make RMI work correctly. In other words, RMI is considered harmful. Our position is that communication within remote processes must be done with asynchronous message passing.

Even though full transparency cannot be achieved, it is important to provide some degree of transparency. We have shown how *port* references and the *send* operation can be used transparently. This is because send works asynchronously and because a broken distributed reference does not raise an exception in Mozart 1.4.0. Instead, a fault stream associated to every remote entity provides monitoring facilities.

We have also described the language abstractions we use to implement Beernet, a peer-to-peer network with a highly dynamic interaction between peers. In order to organize the behaviour of every peer, we have chosen an actor model based on lightweight threads, ports, asynchronous message passing, single-assignment variables and lexical scoping. These language abstractions are very suitable for implementing actors, and they can be used in other programming paradigms.

Acknowledgments

This work is supported by projects SELFMAN, VariBru, and MoVES. The authors would like to thank S. González and P. Hass for fruitful discussion and comments on this work.

6. REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[2] C. Arad and S. Haridi. Practical Protocol Composition, Encapsulation and Sharing in Kompics. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:266–271, 2008.

[3] J. Armstrong. Erlang –- a Survey of the Language and its Industrial Applications. In *INAP'96 –- The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[4] R. Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, Dec. 2007.

[5] R. Collet and P. Van Roy. Failure Handling in a Network-Transparent Distributed Programming Language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'hondt, and W. De Meuter. *Ambient-Oriented Programming in AmbientTalk*. 2006.

[7] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[8] B. Mejías and P. Van Roy. The Relaxed-Ring: A fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, September 2008.

[9] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in E as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[10] Mozart Community. The Mozart-Oz Programming System - *http://www.mozart-oz.org*, 2008.

[11] Programming Languages and Distributed Computing Research Group, UCL. Beernet: pbeer-to-pbeer network - *http://beernet.info.ucl.ac.be*.

[12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[13] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[14] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.

[15] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[16] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1994.