Synchronization & Communication in Mobile Multi Agent Systems, CSP Revisited

Karsten Verelst (kaverels@vub.ac.be)[1]
Werner Van Belle (werner.van.belle@vub.ac.be)[2]
Theo D'Hondt (tjdhondt@vub.ac.be)

Programming Technology Lab (PROG)
Department Informatics (DINF)
Vrije Universiteit Brussel (VUB)
Brussels, Belgium

Abstract

We have created a communication model by extending an imperative language with 4 simple communication primitives. It turns out that these four primitives are very versatile and cover a large portion of existing communication protocols such as CSP and Actor systems.

Our communication model offers both synchronous and asynchronous communication in an easy-to-use communication model. This is achieved by extending an imperative language with 4 communication primitives. Because we desire asynchronous means of communication, our main communication primitives, send and receive, both work asynchronous. We have also included a sync-primitive that synchronizes two processes, and thus synchronous communication can be simulated. The fourth primitive, newagent, is used to create new agents.

In this document we first introduce these communication primitives with the help of some examples. Next we give an example that demonstrates the use of our communication model. This example already gives a hint of the great expressive power of our communication language and in the last part of this document we show that our communication model can simulate the behaviour of more well-known protocols, such as CSP, π-calculus, actor-systems, RMI and others.

Keywords: Distributed systems, communication models, CSP, Pi-calculus, Java RMI

1. Applications of Concurrency

Concurrency is a wide spread fact on the internet and in distributed systems in general. Whenever we have more than one processor and have to write an application which uses services on other machines we encounter concurrency problems. For example, if one uses a CSCW application (say a whiteboard), we see that all parties have to know the internal state of the whiteboard. We also see that these different processes (one on my desk, one on yours) have to communicate updates to each other. How this is done is one of the more important (if not the essence) of the whiteboard application. How we keep the data of all applications in sync with each other is one aspect of concurrency.

Another example of concurrency that we encounter in single processor applications is the interaction between autonomous components, for example between the user interface and the domain application. In such an application the user interface has its own thread which responds to commands such as 'redraw', 'close' and other, while the domain part of the application computes (for example) the state of all players and does the collision detection. How both threads synchronize and obtain both a valid state is a concurrency problem.

In current day distributed systems the implementation of the concurrency is hard wired in the system and is mostly implemented at a very low level. This of course results in messy entangled code which is difficult to manage (and as such is difficult to prove to be correct). For this reason communication models, such as CSP, Pi-calculus, RMI, Corba and IDL's with well suited communication primitives

[1] Karsten Verelst is aspirant at the Belgium Fund for Scientific Investigation (NFWO) and is currently researching security in mobile multi-agent systems.
[2] Werner Van Belle is working at a project for the Belgium Institute for Science and Technology (IWT) and is currently researching software engineering in mobile multi-agent systems.

such as par, alt, guards and others, have been developped.

Nevertheless, if we work in wide area networks in which we have random delay times between processes and in which we didn’t write some of the processes ourselves, we cannot use this kind of transaction system and we have to relay to the interface, the component is offering us, to modify its state. Most of the time this is a synchronous approach in which we wait for the other component to send us an acknowledge. Of course this technique has a lousy performance in wide area distributed systems. Furthermore if the other component doesn’t send an answer at all (because its internals have been changed by the authors) our application may halt because we keep on waiting.

Now, this is exactly the setting which we are thinking of:when we are working in a wide area network in which we have lots of active components (lets call them agents), and which are able to migrate to other places (mobile multi agents). An application consists of fine grained mobile multi agents which interact with each other to obtain a specific goal.

If we ever want to be able to synchronize processes in this kind of systems we need a good underlying fundamental communication model which includes synchronisation between processes. We can think of CSP, pi-calculus, actor systems but it turns out that all these models are only valid in small scaled systems.

2. Problem Statement

Communication models for parallel processes are well understood but their impact on wide area networks isn't well known yet. Most traditional models, like CSP{*} and π-calculus{*}, use a synchronous communication style which are rendered useless in a wide area network due to the large delays that can exist. Other communication models, like actor systems{*}, are already based on asynchronous communication, but “become” difficult to use in practice.

So, looking at current day formalisms we see that they are only valid in down scaled systems in which we own all the processes. To design distributed applications in wide area networks you really need, a formalism that allows

- to synchronize between processes of different owners
- to synchronize in a network with unpredictable delays
- to communicate data to other processes
- to give the concurrency interface to an agent in a clear understandable way
- to create and refer to processes in a simple and understandable way

3. The Cborg Communication Model

Here we outline the requirements our communication model poses on it's base imperative language. We do this by defining a protypical imperative language based on a limited abstract grammar. Actually we impose very little requirements on this language so it can easily be identified with many existing languages like C, Pascal and even Scheme. Even though the language we define is imperative in it's nature, Object Oriented extensions have been developed.

Our language is dynamically typed and a variable can contain values of the following type: numbers, text, tables, functions and references to agents. Variables can be defined using the ``:``-operator, assignment to a variable is written down with a ``:=".

```
a:5
->5
a:=10
->10
```

This is just like in any other dynamically typed language. Actually the communication model doesn't need a dynamical typing at all, so the base language might as well be statically typed.

Equivalent to variables, we can define tables and assign values to any cell of the table.

```
a[5]:0
->[0 0 0 0 0]

a{[3]}:=5
->[0 0 5 0 0]
```

Functions can also be defined using an equivalent intuitive syntax.

```
average(a,b):
        {sum:a+b;
        avg:sum/2;
        avg}

-><function average>
average(4,10)
->7
```

This is about the entire abstract grammar we need for our agent-language. We still need some special language constructs and primitives.

First we need some mechanism to create loops. We have not explicitly added loops to the abstract grammar since we have recursive functions that can execute every possible loop. In the remainder of this document we will sometimes use for-or while-loops, knowing that they can be written as a recursive function.

We also need some kind of if-then-else construction. We have achieved this in our implementation with λ-calculus Church booleans but a syntax-extension is also possible.

Of course we still need a bunch of primitive functions, like arithmetic and string-processing functions, the same that occur in almost any language. There's only one primitive that deserves extra attention. In a system of communicating agents, we have to be able to create references to different agents. The primitive ``agentref'' creates such a reference, specifying the agent by it's name.

```
a:agentref('agent1.localhost.domain')
```

Send

We extend this prototypical language with a pair of communication primitives, send and receive. Send ensures the asynchronous delivery of a message from one agent to another. Because the send command is asynchronous, execution of this instruction results in putting the message in the message delivery system. The call returns immediately and doesn't wait for the message to actually arrive, speeding up program execution by avoiding large delays that can be possible in wide area networks. The downside of asynchronous communication is of course that errors are difficult to detect.

The send instruction needs two parameters to be correctly evaluated, namely the reference to an agent we wish to send the message to, and the message itself. The syntax thus looks something like this:

```
agent2:agentref('agent2.localhost.domain');
send(agent2,'a message');
```

We can send several messages at the same time, or a message with a variable length to some agent, by sending a table containing all messages. This way tagging can also be done.

```
send(agent2,['tag','message1']);
```

Receive

Just as the send, the receive also works asynchronously. This means that the receive is non-blocking, and execution of the program continues immediately, wether there is a message present in the queue or not. The semantics of this instruction is defined to return the message if there's a message waiting to be accepted, or the unique identifier 'nomesg', if no message is present.

agent1	agent2
	recv(agent1,*) :nomesg
send(agent2, 'Kiekeboe')	
	recv(agent1,*) : 'Kiekeboe'

The receive command has several means of messageselection builtin. First of all, the agent, we wish to receive the message from, can be specified. This can be done by naming the sender explicitly, or by using a wildcard, if we're not interested in the sender. If we want to be able to receive a message from a restricted number of agents, we can specify the valid senders in a table. Only messages whose sender appears in the table are returned. If no message from any of these agents is available, nomesg is returned.

recv(agent1, *)	Receives any message from agent1
recv(*, *)	Receives any message from any agent
recv([agent1, agent2], *)	Receives any message originating from agent1 or agent2

The second argument specifies a pattern the message must conform to. This is enforced by a simple pattern matcher that can compare tables, numbers, text and wildcards. In a language where functions are first class, comparison between two functions cannot be added to the pattern matcher, because of non-local variables. Using the pattern-matcher tagging can be accomplished.

Recv(*,*)	receives any kind of message
Recv(*, ['tag', *])	Receives a table that contains 'tag' as it's first element. The second element doesn't matter

Even though the receive command is asynchronous, we can wait for a message to arrive. For now we will show that it is possible. Later on we will show a technique that's more elegant/performant.

```
while(recv(agent1,*)=nomesg) do ;
```

In the language so far it is still impossible to write a server. We can already receive a message from any agent, but we can't reply since we don't always know the senders name. This can be solved by storing the last senders name somewhere.

```
recv(*, *)
send(getLastSender(), 'a reply')
```

Processes

New

Although not really adding to the communicative powers of this model, it is necessary that we include a primitive to create new agents. *newagent* creates a new agent that runs independently and concurrently with it's creator. For this, newagent must know the name for this new agent, together with the instructions it will execute.

```
newagent('helper',
{
parent:agentref('agent1');
result:<some_difficult_computation()>;
ssend(parent,result))
});

helper:agentref('helper');
result:srecv(helper,{*})
```

To facilitate writing code for the new agent, we make the convention that the entire parent-environment is copied. This way, variables and functions can be passed on to the child.

```
foo(a,b):<some_difficult_computation(a,b)>;
newagent('helper',foo(5,8));
```

Note that a copy is made of all functions and variables, and that therefore no concurrency problems can occur between parent and child. Any kind of communication between them must be done with the send/recv primitives.

Sync

The third primitive we've added to our language is called sync and synchronizes two or more agents.

agent1	agent2
sync(agent2)	…
…. Waiting for sync ….	…
….	sync(agent1)
Execution continues	Excecution continues

The parameters for the sync primitive are equivalent to that of the receive. First of all the other agent we wish to sync with must be specified. Again this can be done by explicitly naming the other agent, using a wildcard, or it could be a table with all possible synchronees. The second optional argument is a pattern, putting restrictions on what agents may synchronize. This way we can choose our synchronee with other means than it's name.

Agent1	Agent2
Sync(*,'tag')	Sync(*,'tag')
Continues	Continues

Combining the send, receive and sync primitive, we can simulate a synchronous send and receive:

```
ssend(to,message):
{
send(to,message);
sync(to,message)
}

srecv(from,pattern):
{
sync(from,pattern);
recv(from,pattern)
}
```

Accessing the MessageQ

Asynchronous messages also give us the advantage that we gain some control over the message-queue. Our communication model already has a rather powerfull selection mechanism. However with a clever trick we can peek at messages, edit them, add them and do even more. For this to work we have to slightly edit the message sending protocol:

send(some_agent,[some_agent,message]);

The receive command now uses the pattern-matcher for message-selection to choose between the senders. This way, we can fake message sends and make the interpreter believe the messages were send by another agent. This behavior comes in handy when we want to reinsert messages in the queue. We just fake-send the message ourself and later on we will not be able to tell the difference between an original and a reinserted message.

The principle of accessing the meta-queue now becomes simple. We dequeue the message and we reinsert it back into the queue. Now we have full random-access control over the meta-queue. We can insert, remove and peek at any message in the queue.

Because it mght be difficult to change every message send and receive in an entire program, we suggest that any implementation includes some queue-accessing primitives. Note that the programming technique presented above only works with asynchronous messages, not for the sync primitive.

4. Whiteboard Systems

As an illustration of the communicative powers of our model we will give a standard example of a cooperative whiteboard in which members can join to an existing cluster. Receive updates and are able to send updates. The whiteboard itself is used to draw figures on. Whenever a client is changing a figure (a line, a circle, a box) for example, all clients will show this figure in red until it is released again.

The weak points in such a design is the ability to synchonize between clients to lock the given resources. Normally we send a lock-request to everybody. All clients will 'almost-lock' this recoures and send an acknowledge (or an already-requested-for-lock message). At the moment we are unable to lock all clients we will release all locks and try again later. If we have almost locked all clients we lock all clients for real this time and the colors and the behaviour of the objects can change.

In a normal programming language this form of optimistic locking requires a large number of code lines to implement. If we use our formalism and language extentions we can define an interface for a client as follows:

```
sync(with,['lock'],clientnr, objectref)
```

after the sync request this objects and this client will continue with a state in which the given objectreference is locked by the client clientnr.

```
UpdateState(objectref,data)
```
Sends an update to another client with the new state of the object

```
GetObjectList()
```
Receives the list of objects which are shared by all clients.

```
GetState(Objectref)
```
Gets the state of the object referenced by objectreference.

```
Connect()
```
Connnects the sending client to the receiving one, so updates will also be propagated to this client. Every connecting client has to connect to all other connencted clients.

From this example we see that our formalism has the very strong advantage to be able to give a standard syncing interface which declares how the state of the application will be after the synchronisation. The sync itself takes care of waiting for all clients while the code after the 'sync' can immediatelly lock the given resources.

5. Other Communication Protocols

CSP

CSP is a well-known language used in concurrent systems. It is based on a simple imperative language and 5 basic commands: send, receive, par, alt and guards. Communication in CSP (send/recv) is synchronous and both primitives must specify the other process explicitely.

The par command simply executes a number of processes in parallel and all these concurrent processes share the same parent-environment.

A guard consist of a condition (head) and an expression. The expression will only be executed once the head evaluates to true. Because the head can contain synchronous communication, we don't know when and if the head will become true.

The alt-command consist of a number of these guards, but only one of all guards that might become true, will be evaluated.

CSP is a very powerfull language for closed concurrent systems, but isn't adaptable to work in distributed systems. Amongst others, the constraint that every communication primitive must specify its partner explicitely, forms a huge barrier in porting CSP to WAN's.

It can be shown that all CSP-commands can be expressed in our communication model, thus giving our model at least the same expressiveness as CSP.

Π-calculus

Robin Millner's π-calculus is a simple calculus based on naming. π-calculus uses channels, identified by names as their primary communication primitive. The calculus itself consist of a number of simple expressions like: 0 (the empty process), if, par, send, receive, restriction.

Again the send and receive are synchronous and will only communicate if both of them specify the same channel to communicate through. A restriction introduces scope into π-calculus. With such a restriction, a names scope is restricted to the adhering code.

Because π-calculus doesn't constrain what may be communicated through the channels, chunks of code (actually references to them) can be sent, making the π-calculus the first calculus which supports the notion of mobility.

Our experience with Robin Millners π-calculus has tought us that π-calculus is too low-level to use in practice. Extensions have been built to remedy this problem, yet they didn't quite succeed.

In our protocol, a π-calculus program can be simulated. What we actually do is replace each name by a node. Each node contains a reference to the nodes formed later in the program and communication through a named channel can then take place by informing the channel (node) of the message we wish to send/receive.

By implementing the π-calculus we automatically inherit the same principle of mobility and every other feature that π-calculus offers.

RMI

Remote Method Invocation is a protocol added to Java that makes it possible to invoke methods on remote objects. The method calls are synchronous, which makes it sometimes difficult to use them (e.g. returning the value of computation later on), but this protocol is rather popular in distributed circles (along with CORBA). Again it can be easily expressed in our communication model.

Actor Systems

Actor Systems are another communication model based on λ-calculus extended with a couple of instructions. Of course there's an instruction that creates new agents. There's also an asynchronous send method included, but no explicit receive. A send to an agent results in the application of that agent (λ-expression) with the received parameters. Actually we have a kind of asynchronous procedure call.

The last instruction and the most difficult instruction is the *become*, that changes the behaviour of an agent, letting an anonymous agent carry out the current computation.

Every instruction (except the become) can be trivially stated in our communication model. To implement the become, we create a forwarding proxy for every agent. This way we can easily update the proxy, to point to a new agent we just became.

By comparing our communication model with some others (CSP, π-calculus, RMI and actor systems) and each time showing that our model has the same behaviour, we have proven that our model has at least the same expressiveness as any of these other protocols.

6. Cborg

We have implemented this communication model in a state-of-the-art mobile multi-agent system. The agent system is based on an educational language, called Pico. The system itself supports location transparent routing, strong migration (at any time can we move an agent, even if it's in the middle of the execution of code) and a very suitable commuication paradigm. More information can be found at http://progwww.vub.ac.be/pools/cborg. Furthermore, we have implemented CSP and the PI-calculus within our system.

7. Future Work

If we work in wide area distributed systems time outs are a rather important concept. So in the near future we may have a look at how these can be easily integrated in our system without messing up the complete formalism.

8. Conclusion

We have created a new communication model by extending an imperative language with four communication primitives: send, recv, sync and newagent. The send and receive both work asynchronous meaning that execution continues without waiting. Therefore the receive returns the unique nomesg identifier when no messages can be received or the message itself if there's one in the queue. The sync, synchronizes two agents and newagent creates a new agent. To make it easy to work with new agents, the parent environment is entirely copied, making it easier to work with these new agents.

After we have defined the communication model, we have given some examples of what can be done with it. We have first shown that it is possible to implement a kind of distributed Object Oriented system, and then we have seen how we can gain random access control over the meta queue. Especially the meta-queue control routine gives great power and flexibility to our communication model.

Finally we have proven that our communication model exhibits the same expressive power as any of the better known communication models.

We don't claim our approach will solve all 'transaction' problems, but it will nevertheless be much more easy to implement transaction systems and try other approaches to ease the concurrency burden.

9. References

[1]*C.A.R. Hoare,* Communicating Sequential Processes, Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[2] *Milner, Parrow and Walker*, A Calculus of Mobile Processes Pt.1 LFCS report ECS-LFCS-89-85

[3] *Milner, Parrow and Walker* A Calculus of Mobile Processes Pt.2 LFCS report ECS-LFCS-89-86

[4] *Robin Milner* The Polyadic pi-Calculus: A Tutorial LFCS report ECS-LFCS-91-180

[5] www.pico.vub.ac.be

[6] http://prog.vub.ac.be/poolresearch/CBorg/

[7] *Gul Agha, Ian A Mason, Scott F Smith, Carolyn Talcott,* a Foundation for Actor Computation, Cambridge University Press 1993