Is Message Sending Good Enough For Distributed Systems ? Synchronization & Communication Revisited

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

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

This position paper looks at current day distributed object systems. In most of these systems we see a basic building primitive which offers us the needed distribution capabilities. Most of the time this is a synchronous or asynchronous message send. We argue that this is not good enough for distributed systems. Choosing the 'message−send' as primitive operation creates too many problems and creates the illusion that writing distributed programs is as easy as 'calling an object'. This is not true. In this paper we would like to present our ideas about this and hope to provoke some discussion about them.

Section 1: Distributed Systems

Many large applications today are built using Object Orientation. This is because OO allows the programmer to think in a more abstract way about his code. OO enables him to encapsulate data, with *private*, *protected* and *public* fields, and use the implicit notion of *this* (or *self*). The programmer is now able to request a service from another object by sending a message to it. He has even the power to create something with holes, which will be filled in later (late binding, polymorphism, the *abstract* keyword). These techniques are very useful, and have changed over time to adapt to the needs of programmers. For example, *interfaces*, as defined in java, are needed to treat different object hierarchies as if they were the same. And over time some concurrency primitives appeared (like *synchronized*).

Along the same way we see people experimenting with Distributed Systems, adapting the OO paradigm to make the writing of distributed programs easy. Actually, there is a tendency to make everything transparent. Of course, the complexity of the problem is often completely neglected. Writing distributed programs is *much* harder than writing conventional Object Oriented applications. We have for example,

- Delay Times: Transmission time with other parts of the systems, which reside somewhere else, is much higher (let's say 100'000 times at least) in comparison to transmission times within the same computer.
- Concurrency: distributed systems connect computers to each other, so we have an implicit parallelism which, whatever we do, will pop up in our applications.
- Ownership of Data: parts of distributed systems are written by other people, and can be changed while we are using them. We cannot rely solely on the interfaces we are using.
- Partial Failure: any part of the system can go down at any time for no reason at all. Sometimes failure is not even reported and the system just doesn't respond.

These problems are hard, in the sense that we *can not* easily abstract them. This is also the reason why transparent distributed systems are doomed to fail. When writing a distributed program we have to *think* in another way and not act as if there is no distribution involved. E.g.: taking an existing application and making it distributed. If you do this (or have done already) you will see that you have to rethink/redesign the complete application, even within a suitable distributed OO language.

So where does this leave us ? We cannot make the given problems (concurrency, partial failure and so on) transparent, but we also cannot expect the programmer to put the bits on the wire himself. We have to find something in between them, something which offers the programmer a well defined, useful primitive distribution operation. For many people, this primitive is the 'message send', we will try to show that this may not be a good choice, whether the message send is synchronous or asynchronous. We will try to give some guidelines of what we consider better primitive operations, and hope to provoke discussion about them.

[1] 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.

[2] Karsten Verelst is aspirant at the Belgium Fund for Scientific Investigation (NFWO) and is currently researching security in mobile multi−agent systems.

Section 2: The perils of the Synchronous Message Send

First lets first take a look at the synchronous message send of Java. In Java we can send a message to another remote object by sending the message to a local stub. This stub will call the remote object, wait for an answer and return. In practice this means that we can write code like this

```
a.joinWhiteboard("Werner","26")
```

Where *a* is a local reference to the remote object. The call itself will wait until everything has worked out, or throw an exception if some kind of (network) error occurred.

2.1 What to pass to the other side ?

If we use this easy mechanism we immediately have to decide what to pass to the remote side. Java allows us to serialize objects and object graphs, but serializing and transmitting everything is maybe a bit too much. E.g.: a whiteboard client which joins itself to a whiteboard can pass itself as a parameter.

```
a.joinWhiteboard(clientapplication)
```

Eventually, we end up transmitting everything, that is, the complete client application. Luckily, Java allows us to specify which objects can be serialized (*serialisable* interface) and how certain objects should be serialized (*externalizable* interface).

This small example illustrates that a simple message send is in fact not that simple any more. If we start using it we have to *think* of what should reside where.[3]

2.2 Message Sends can Fail

Another problem with synchronous message sends are failures. Every call we make can fail with or without notification. In Java we are lucky to have the *remoteException* exception, which we can either catch or throw ourself. E.g.

```
try {a.joinWhiteboard(ClientApplication)}
        catch (AccessException e) {...}              // cannot access remote object
        catch (ActivateFailedException e) {...},     // oops, it was asleep and didn't wake up
        catch (ConnectException e) {...},            // oops, cannot connect
        catch (ConnectIOException e) {...},          // oops, transmission errors
        catch (ExportException e) {...},             // oops, programming error at this side
        catch (NoSuchObjectException e) {...},       // oops, remote object has died
        catch (ServerError e) {...},                 // some logic error occurred at server
        catch (ServerException e) {...},             // oops, server threw an exception
        catch (ServerRuntimeException e) {...},      // oops, server died when handling message
        catch (UnknownHostException e) {...},        // oops, host unknown
        catch (UnexpectedException e) {...};         // oh dear, something else happened
```

Well.. 'Lucky to have a remoteException'... look at this bunch of possibilities. If we have to check every possible message or sequence of messages for these kind of errors, our code will be barely readable, and the simple 'just send a message' concept is simple no more. Of course, whether we do this in Java or any other language, every possible message send can fail, which makes it impossible to write code in an understandable way. *Treating remote objects as local objects: 'just call them' doesn't work.*

2.3 It's not asynchronous

The last neglected major problem of the synchronous message send we will describe is it total lack of performance. Running a program just as if it is a single–threaded application and wait at every message send for an answer from the other side, *while it might not be necessary*, slows down your application too much: You don't want to wait for a dead process. Sometimes (most of the time, we dare to argue) programmers of distributed systems will want to use asynchronous messages.

Now, we would like to wrap up the description of the synchronous message send by concluding that it is a nice construction to test *small* distributed programs, but hard to use in practice because it tries to hide too much. Now, let us have a look at the asynchronous message send.

[3] The people at Sun had to add the *Serialisable* interface as a means to make the programmer *think* of what he is doing. It would have been perfectly possible to make everything serialisable and only to offer a externalizable interface where the standard behavior wasn't good enough.

Section 3: The ’features’ of the Asynchronous Message Send

Let’s say that we have an asynchronous message send resembling the Java synchronous message send, but now it doesn’t throw errors or wait[4]. In fact, this message send doesn’t return anything at all. E.g.:

```
a->joinWhiteboard(clientapplication)
```
[5]

will send the message *joinWhiteboard* to the other side. To return an answer to the client, the server will call a method on the client. For example, the message *whiteboardJoined*(). It is obvious that the client needs to provide/implement this callback interface.

```
void whiteboardJoined(int state) { ... change state ... }
```

3.1 Writing State Based Programs

If there is only such a kind of operation available, we immediately see how difficult it is to write simple programs. We now have to distribute the computation over a number of communication partners. ’I calculate something, you do something else, and after that I will continue again; if you didn’t answer in time I will do something else, but if you answered too late I still might use your result, if ....’.

Writing down such an interface requires a disciplined use of state machines. It is no longer feasible to expect a straight−forward message sequence from another program and rely on it. Or even worse: make your implementation dependent on a certain sequence of messages. We are now forced to write programs as state machines with state transitions partly initiated by other programs. This implies that we have to explicitly take into account what happens when errors occur. Programs implemented this way become reactive to their environment instead of passive entities which expect a certain behavior of other programs.

So, the major advantage of using an asynchronous send is that it coaxes the programmer to write his programs in a state based way. He now has to *think* about the logic of the application: what will happen when this or this message arrives at this moment ? He cannot ’hide’ errors and message arrival in the execution flow of the program. He now has to *integrate* them into the program. Which is, admittedly, difficult, but necessary.

Nevertheless, asynchronous message sends also have some drawbacks, as illustrated below.

3.2 How do we know what we were doing ?

Suppose we have a webserver process written as follows:

```
...
HandleSocket(Socket s)
        {
        Url u=s.readUrl();
        Remote h = getUrlHandler(u);
        h->generateHtml(u);
        }
...
```

All programs which can generate Html can subscribe to a certain Url on the webserver. *HandleSocket* is called whenever a web browser connects to this webserver. If this happens, the webserver will look up the program which will handle the Url and send a *generateHtml* message to it. The handler program will normally send a *Html* message back, containing a string. The problem we are facing, is how to write the *Html* function in our webserver. It should do something like:

```
Html(String html)
        {... send string over the previous socket (?)...}
```

The program should be able to ’remember’ the initial request from which socket it came at the moment the *Html* message arrives, which makes writing programs in a state based way a bit more difficult.

This suggests that there are still some problems with asynchronous message sends, which we will investigate in the next section ’What about Message Arrival ?’.

---

[4]This is similar to *Actor* Asynchronism

[5]The −> notation is used to specify an asynchronous message send.

Section 4: What About Message Arrival ?

We looked at the synchronous message send, which created the illusion that writing a distributed application is as easy as sending a message. Afterwards we investigated the asynchronous message send, which turned out to force the programmer to think about his application in a state based way, but made implementing the software tricky because we are unable to receive messages in a decent way. Now, before we take a look at synchronization we take a sidestep at an interesting asymmetry in distributed object systems.

4.1 Sending To vs Receiving From

Using an asynchronous delivery system, it is easy to send messages to all our clients. E.g.: the Whiteboard server can send an *okToRestart?* message to all its clients using a simple loop.

```
for(int i=0;i<clients.length;i++)
        clients[i]->okToRestart(someargs)
```

Broadcasting *okToRestart* is no problem, the problem occurs whenever one has to wait for all the clients. This is difficult to achieve. If we want to receive from all the clients an 'okRestart' message, we end up writing quite complex code like this:

```
void okRestart(ClientProtocol from)
        {
        if (clientsRestart.waiting()) return;
        if (clientsRestart.contains(from)) return;
        clientsRestart.add(from);
        if (clientsRestart.allOk()) {... change state ...}
        }
```

The problem itself is quite obvious and fundamental according to us: Object Orientation is unbalanced in respect to *send* and *receive*. *Sends* can be scattered throughout the code, while *receives* are only possible at method declarations. As such distributed object oriented programs are forward driven, they are pushed by the actions the programmer wants to take and are unable to deal with changing situations. We are faced with a language paradigm unable to offer a *dynamic receiving behavior.*

4.2. How to offer a Dynamic Receive ?

A possible solution to this problem is to offer a special message receive. A receive that checks whether there is an incoming message in the incoming message queue that matches a given pattern. This message receive can be placed at E.g.

```
someclient<-join(?clientid)
```

will look for a message *join* with one parameter coming from *someclient*. If such a message is available the first matching message will be removed from the queue and the variable *clientid* will be assigned the value which has been sent to us. If we would write

```
?someclient<-join(?clientid)
```

we would look for any message, coming from anywhere which requests for a join. The first matching message found will be returned and the variables *someclient* and *clientid* will be filled in according to the this message. The question remaining is what happens when no such message is available. Is it appropriate or not ? Waiting would be nice, but it has the same disadvantages as mentioned with the synchronous message send. So, it is maybe a good idea to choose the receive to be asynchronous.[6]

4.3. Does it Work ?

Does this explicit receive construction work ? Maybe, have a look at the previous example. Sending was easy,

```
for(int i=0;i<clients.length;i++)
        clients[i]->okToRestart(someargs)
```

and receiving looks easy too.

```
for(int i=0;i<clients.length;i++)
        clients[i]<-okRestart(?someargs)
```

This would work if the receive was synchronous, the problem is that it isn't. So this might look a nice construction, but we are *still* missing some quite fundamental functionality. We are unable to *wait* for other processes, to check which messages are available and to act upon this.

[6]The result of the <– operator is *true* if such a message is available and *false* if none is available.

Section 5: A little thought experiment: Synchronization

In our thought experiment we have a dynamic asynchronous send and a dynamic asynchronous receive, nevertheless its all a bit too asynchronous. We have no way (except polling) to synchronize. It's time to take a look at a fundamental paper about it: Hoare's CSP.

5.1 Guarding and CSP

CSP [1] is a well−known language used in concurrent systems. It is based on a simple imperative language and has 2 small sets of basic commands. The first set are primitives for communication: *send* and *receive*. Communication in CSP is synchronous and both primitives must specify the other process explicitly, which immediately renders them useless in error−prone distributed environments.

The second set of primitives are those for synchronization: *alt, par* and *guards*. The *par* command simply executes a number of processes in parallel. A *guard* consist of a condition (head) and an expression. The expression will only be executed once the head evaluates to true. The *alt*−command consists of a number of these guards, but only one of all guards that might become true will be evaluated.

The interesting thing about these *guards* are that they are explicitly written down and consist of some conditional code which is managed together with other guards. If we write a guard in an alt statement we know that executing the body will only be done when nothing else is happening *in the alt* statement.

So where does this lead us to? We know from literature and experience that we need some kind of synchronization management (*alt/par/seq* in our case) and we also need some kind of synchronization. In CSP this is easily done, by placing a rendez−vous between a *send* and a *recv*. In our little experiment, we have an asynchronous send and an asynchronous receive, which requires us to offer an *explicit* synchronization primitive.

5.2 Synchronization

Synchronization means that we wait for a certain condition to happen before we continue. The condition itself should be stated as clearly as possible with no side effects, like, for example, a 1st order logic predicate.

Furthermore, synchronization has some requirements which should be taken into account:

- In distributed systems, the synchronization mechanism should be implemented locally. It should not need to communicate with other partners to synchronize, because communication failures can endanger correct working.
- Synchronization should always time−out, so we can take appropriate action when communication failures happen.
- *Synchronization should not depend on the behavior of other threads. This means that all incoming messages are offered to all waiting synchronization points and that all these threads can pop the given message from the queue. Every thread should be able to retrieve all incoming messages.*
- *A good synchronization primitive should be clear and have no 'dirty', unexpected behavior: Between synchronizing at a certain kind of messages and looking at the messages there should be no unexpected message arrival. In other words: between two sync operations we should have a view at a seemingly frozen queue.*

We will denotate such a synchronization operator with ~. Below are some self−explanatory examples of the ~ operator.

```
~(A<-okRestart)        // wait for an okRestart message from process A
~(?<-okRestart)        // wait for an okRestart message from anybody
~(A<-?)                // wait for any message coming from A
~(?<-?)                // just wait for any message to occur
```

In the following sections we illustrate this operator

5.3 Waiting for all clients to join

The following example illustrates how we can be in a state where we are waiting for clients to join or in a state where we are waiting for the start signal from the gamehost.

```
do {
   ~(?<-join || gamehost<-start)             // waiting for start signal or join request
   while (?client<-join(?name))              // handle all pending joins
        {
        client->joined(motd)                 // send joined message to client with motd
        System.out.println(name+" has joined");
        }
   }
until (gamehost<-start)                      // repeat until the gamehost wants to start
```

5.4. Waiting for other processes before sending

In the same way as above, we can use the –> operator in a sync statement. This means that de operation will sync at the moment the given message has been sent by another process. E.g.:

```
par(
  p1: {...                    // process 1 does something
      b.act(15)               // send the act message to b
      ...}                    // process 1 continue
  p2: {...                    // process 2 does something
      c.act(12)               // send the act message to c
      ...}                    // process 2 continues
  p3: {...                    // process 3 does something
      ~(b->act && c->act);    // process 3 waits for process 1 and 2 to send the message
      d.act(9)                // and send act to process d
      ...}                    // process 3 continues
```

We see in this example how we can wait for other processes to send something out before we send something to another partner.

6. Meta Object Protocols ?

Its clear now that we need other operations than an object oriented message send to program distributed systems. But what kind of operations do we need ? What about are own synchronization experiment ? It's nice, but is it the best ? We don't think so. We are certain that somebody else will find better operations for his (or here) problems and claim his approach to be the best. So, why not use meta object programming ?

The problem with Meta Object Protocols is that you can do everything with it. You don't like the message send ? Well, you can redefine it. In the end everybody writes their own primitives and nobody actually understands anything about the semantics of the message send. So, we don't want people to change a basic primitive operations *always*, because that would mean that our primitive operation was not good enough. It would be better to have *good* operations available, which *can* be changed when necessary, but shouldn't most of the time.

A second problem with meta object protocols is that they tend to adapt semantics of existing operators where it is not suitable. *As* illustrated in this paper, message sending is not the right operator for communication with other processes. So why would we even *try* to change the semantics of the message send. What we need is something else, some new operator which has nothing to do with the former. Meta object programming is a good at changing the semantics of certain operations *slightly*. It's not suitable for introducing complete new semantics into a programming language.

Conclusion

This paper argues that a synchronous message send – despite of how appealing it might be – is not a very good choice for implementing distributed systems. An asynchronous send is definitely much better, because it requires the programmer to *think* in a state based fashion. However, this operator makes synchronization between processes too difficult to be useful. Therefore we created a suitable synchronization operator to fit our needs.

In this paper our goal was to stress the importance of good communication and synchronization primitives. Only adapting a standard object oriented paradigm to *hide* the complexity of distributed programs is an illusion. Therefore we should aim at developing a good set of useful and flexible primitives, like, for instance in this paper: an asynchronous sending operation, an asynchronous receive operation and a synchronization operation.

Acknowledgments

We would like to thank Bart Wouters, Dirk Van Deun, Johan Fabry and Tom Tourwe for their patience in reading the drafts of this position paper. This position paper is based on the graduation thesis of Karsten Verelst and also contains parts of brainstorm sessions with Kristof Van Buggenhout.

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] *Robin Milner* The Polyadic pi–Calculus: A Tutorial LFCS report ECS–LFCS–91–180
[4] http://pico.vub.ac.be/
[5] *Gul Agha, Ian A Mason, Scott F Smith, Carolyn Talcott,* a Foundation for Actor Computation, Cambridge University Press 1993
[6] http://borg.vub.ac.be/; http://borg.rave.org/; http://progpc26.vub.ac.be/
[7] The Java Object Serialisation Specification