Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System

Werner Van Belle
Werner.Van.Belle@vub.ac.be

Johan Fabry*
Johan.Fabry@vub.ac.be

Karsten Verelst†
Karsten.Verelst@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

http://prog.vub.ac.be/CBorg
Vrije Universiteit Brussel - Programming Technology Lab
Pleinlaan 2, 1050 Brussel - Belgium

Abstract

This paper reports on our experiences in the field of mobile components. In the past 4 years we developed a mobile component system, which allowed us to experiment with code mobility in distributed systems. These experiments have given us a unique opportunity to study two major issues in mobile component systems. The first issue is how to develop and provide a robust mobile component architecture. The second issue is how to write code in these kinds of systems. This paper discusses our experience in both of the above.

1: Introduction

About four years ago, the Programming Technology Lab of the Vrije Universiteit Brussel initiated research in the field of mobile computing. More specifically, a number of researchers have been implementing a mobile multi-agent system, as defined below:

A mobile multi-agent is an active autonomous software component that is able to communicate with other agents; the term mobile refers to the fact that an agent can migrate to other agent systems, thereby carrying its program code and data along with itself. Regarding the terminology *mobile multi-agent system* there is some confusion. A multi-agent system in AI denotes a software system that simulates the behavior of large groups of interacting agents (called multi-agents), without focusing on the distribution aspect of these systems. In the world of distributed computing, a mobile multi-agent system is a *distributed* environment in which multi-agents can be written. We adopt the second definition.

The system we built, called CBorg, provides a platform for conducting experiments with active autonomous agents, which communicate with each other over a wide-area network, and which are able to migrate over this network. The CBorg agents can be considered as mobile components: a component is an active piece of code, which can communicate with other components on the network. A component is able to migrate to other machines. A

*Author funded by a doctoral grant of the Flemish Institute for the advancement of scientific-technological research in the industry (IWT)

†Author is a Research Assistant of the Fund for Scientific Research Flanders (Belgium) (F.W.O.)

component's state can only be modified by sending a message to that component; all data of a component is private. In the remainder of the paper we use both terms (component and agent) interchangeably.

Note that when we say "component", we do **not** mean "object". Because objects share a data space and are passive they should not be considered as being components. Components are active entities with independent private data. Usually a component consists out of a number of objects.

In this experience report, we discuss some of our design decisions and offer a guide to implementing mobile components. The paper is structured as follows: first we give an overview of the application domain in which we envision the use of mobile components, and describe the problems encountered while using existing infrastructures to implement applications in this domain. Subsequently we introduce CBorg: the experimental infrastructure we built to address these problems. The final section discusses how mobile components should be written using this infrastructure.

2: Application Domains

One of the first advertised applications of mobile agents lies within the field of E-commerce. In this setting, agent technology would help the user when purchasing certain goods [2]. Consider a pricing agent, which helps the user to obtain the lowest possible price for a given good. Let us imagine that, as an example, the user wishes to buy an mp3 player. The agent, which is located on the user's machine, will request the specifications the player should have, e.g. the number of songs it can contain and a maximum price. Once the specifications are gathered, the agent will migrate itself towards different known vendors of such players, and at each vendors' location request the prices of mp3 players matching the specifications. When all vendors have been visited, the agent will return to the user, and at this point it will present the information it has gathered. A number of so-called shopping bots are already available on the Internet, but they do not use mobile agents for collecting their data.

Another interesting field of literally mobile computing, is the field of hand-held computers, such as the Palm$^{\mathrm{TM}}$. These devices are truly mobile, since a user will carry them around wherever she will go. According to the location the user is in, she will need different applications and/or different data. Since a significant property of hand-helds is their limited amount of resources, such as memory, it is awkward to continuously keep all needed applications on the hand-held. For example, the user would need a city map application only when visiting a foreign city. When not travelling, such an application and its large amount of map data only uses up valuable space in the computer's memory. With current-day systems the user needs to explicitly install the required software on the hand-held before embarking on a trip, and uninstall it when memory is scarce. With a true world-wide mobile computing infrastructure, the hand-held would detect that it is at a location foreign to the user. It could then automatically migrate a city map application from the user's desktop computer to the hand-held, and possibly off-load other, unnecessary applications, such as an electronic book, to the user's desktop computer to preserve memory. The inverse would happen when returning to the user's home city; the city map application would be removed from memory, and the electronic book would be migrated back to the hand-held.

A second application of mobile computing in hand-held computers is the off-loading of computing from the hand-held to a fixed computer. As stated above, hand-helds have a limited amount of resources, for example, a slow CPU and a short battery life. We can preserve resource usage, thereby extending battery life and possibly even speeding up response time, by moving a resource-intensive agent off the hand-held to a more powerful and less resource-constrained machine. An example of such a resource-intensive agent would be an agent that performs a large number of queries over the network, and performs a significant amount of computing, such as the shopping agent we described above.

A somewhat more innovative field would be to deal with roaming users. Roaming users are users who do not have one fixed computer, or a fixed office, from which they work, but who will use any one of a set of computers, usually a different computer for each working session. The task is to provide the users with a consistent desktop across the different machines, and to allow a user to seamlessly suspend a session on one machine, and resume the session on another machine. This is more powerful than simply closing all open applications and logging off. It should be possible to simply halt all applications, saving their complete state, and resume from that point later on, just like pressing a 'pause' button on a VCR. If we would want to allow for seamless suspension and resuming of a user's session without using mobile agents, we would need to run all computation on one central server. Pausing a user's session would be achieved by writing the user's state (equivalent to the content of all memory allocated by the user) to disk. Resuming is achieved by restoring the user's state from the disk. However, running all computation from a central server is very expensive. It requires a high-performance server, to be able to handle multiple user's sessions simultaneously, and a network with a large bandwidth, to transmit screen updates to the different users in a timely fashion. It would be more efficient to let the computation run locally, which allows for cheaper hardware (on a per-user basis), and does not require a high-speed network. When suspending a session, we can just suspend the agents and move them to a storage server. This storage server will keep the agents in permanent storage. Whenever the user logs in, the agents will be moved to the computer the user uses for that session. An interesting extra feature here, analogous to the existing field of application servers, is the ease with which software can be updated. Whenever a new version of a mobile agent is released, this agent need only be "installed" on the storage server.

Of course, writing such mobile agents in a current-day system is not a trivial task. One of the main reasons for this difficulty is that we do not have a worldwide homogeneous architecture for mobile agents. For example, it is impossible to run code for a Motorola processor on an Intel processor. The Java VM offered a basic solution for this problem: it is now possible to move code from one computer to another with the certainty that the code will run. Nevertheless Java still lacks important support for mobility of running programs.

Suppose we try to create mobile agents with Java; we would probably write a small Java program using sockets to connect to other Java-agents. The first problem we encounter is how to find the other agents and how to keep connected to them when we migrate. Moreover, migration itself also turns out to be a significant problem. While moving Java-programs is easy — just send the class over and run it — migrating running Java programs is a lot more problematic. This is because we are unable to capture the execution state of a process, i.e. we can neither serialize the Java execution stack, nor serialize a thread.

Another reason for the present-day scarcity of mobile agents is that the code is not written to be autonomous in a dynamic environment. For example; when writing OO software we send messages to other objects, with the certainty that they will be executed immediately and that we will get an answer. This is no longer true in open distributed systems, where we have large delays in message delivery, and where the remote computer may fail.

3: The CBorg Infrastructure

To allow us to easily experiment with mobility in a distributed environment, we built the CBorg platform, which could be considered as a basic infrastructure for worldwide mobile computing.

CBorg agents ('Migrobs') are written in a high level Scheme-like programming language, called Pico [3]. Pico is an educational language used at the Vrije Universiteit Brussel to teach concepts of programming languages to first-grade computer science students. Small and portable by design, Pico allows us to experiment with new concepts and language constructs for mobile multi-agents[1]. On top of Pico we have built an integrated development environment (IDE) that supports the interactive writing and designing of agents.

When we start the CBorg interpreter, it connects to another CBorg interpreter, which yields an interconnected network of agent systems. This interconnection of mobile multi-agent systems is called the mobile multi-agent infrastructure.

Currently, the CBorg mobile multi-agent architecture features:

- **Strong mobility**, meaning that an agent can migrate between agent systems while it is executing. Strong mobility is seldom found in current, Java-based, agent systems due to some technical drawbacks of Java. Because CBorg has the ability to reify the complete computational state of a running process, including its runtime stack, strong mobility is one of its standard features. Furthermore we have the ability to store and retrieve computations as variables (**continuations**) and pass these to other agents (**remote continuations**).
- An easy to use **agent communication layer**. This communication layer, which consists of a serializer and an objectcall-like syntax, allows agents to pass messages to each other. Agents always communicate in an asynchronous fashion. The reasoning behind this design decision is the notion of being 'autonomous': an autonomous agent should be designed as a separate entity, sending messages to, and receiving messages from other agents, not as an entity which transfers its control flow to other agents.
- **A hierarchical naming system**: every agent has a human-readable name, which is always used to reference it. The naming system favors late binding, in the sense that we bind agents to each other at execution time, not at compile time, as we partially do with objects.
- **A high-performance location-transparent distribution layer**: an agent can send messages to other agents, without having to know where the other agent resides. For example, if agent 'Alice' talks to agent 'Bob', and 'Bob' migrates to the agent system at the end of the universe, CBorg keeps on routing messages between Bob and Alice using the shortest path between them.
- **Resource Transparency**: all resources in the mobile multi-agent system (disks, user interfaces and so on) are represented as static agents (which cannot migrate). So

[1]Because of the experimental nature of CBorg, we do not have any intention to be compatible with existing languages, such as Java.

whenever we migrate an intelligent agent, it stays connected to the resources it was using.

- **Garbage Collection**: we have a state of the art, highly performant garbage collector incorporated into the system. The garbage collector uses a 32-bit cell memory, with only 2 bits reserved for bookkeeping!
- **Synchronizing agents** is performed by using a rendez-vous between multiple agents. This rendez-vous can be in time and/or in space (synchronize at a certain computer). The primitives themselves are based upon CSP [5], with the exception that we use unification instead of sequenced statements as guards.

We now discuss four of the above items: the location-transparent distribution layer, strong mobility, synchronizing agents, and agent communication.

As an illustration we use the example of a web server throughout the discussion. The web server consists of a varying number of agents, depending on the required functionality. The main agent is called WebDispatcher, it dispatches incoming requests to appropriate secondary agents, which generate HTML for the given request. Incoming requests are transformed into a WebRequest agent, which announces itself to the WebDispatcher. The actual CGI-like processes (UrlHandlerAgents) announce themselves to the WebDispatcher by subscribing on a number of URLs.

3.1: Location Transparent Distribution Layer

One of the most immediate problems we encountered was the total lack of support for mobility in existing distribution protocols. For example: if we implement an UrlHandler which can migrate from one machine to another, how does the WebDispatcher communicate with this 'jumpingUrlHandler'? A common solution to this is to let the programmer of the WebDispatcher solve the problem of localizing the UrlHandler and managing a communications channel to this agent. This solution is not acceptable because the problem can be solved by the distribution layer itself.

We wanted a distribution layer that was able to name agents uniquely, worldwide (for example, a counter URLHandler called `prog/wernersCounter`), in a location-transparent fashion, while taking mobility into account (if the agent migrates to another machine, its name should remain the same). This naming system should be used to reference agents and to send messages to them. To provide this functionality, we merged name server and router into one entity.

3.1.1: Sending Messages

The solution we implemented is based on abolishing the distinction between the name of an agent and the address of an agent. Instead of resolving the name of an agent to find its place, we immediately route messages to an agent based upon the receiver's name. This means, of course, that we need to substantially change the existing communication infrastructure. We no longer have a statically interconnected routing infrastructure and a separate, statically interconnected naming infrastructure; instead we have one hierarchical infrastructure in which we name agents and route messages between them.

To send a message we do not send a lookup request to a name server, but send the complete message to the name server, which 'routes' the messages further to the next name server. For example: the figure below contains a hierarchical interconnection of name servers/routers.

If the agent `prog.vub.ac.be/WebDispatcher` wants to send a message to an agent `belnet.ac.be/WernersUrlHandler`, it can pass the message to the local name server, which is in this case `vub.ac.be`. At the moment `vub.ac.be` receives a message, it sends the message through to `ac.be`. Then `ac.be` sees the message for `belnet.ac.be/WernersUrlHandler` and passes it to the right downlink, in this case, `belnet.ac.be`. There it is immediately delivered to `belnet.ac.be/WernersUrlHandler`.

3.1.2: Migrating Agents

Migrating an agent in this setup is similar to sending a message, with the difference that every router/name server should interpret the movement of an agent. This means that whenever an agent passes through a node, the node should update its routing-tables to point to the agent's new direction. Note that we do not point to the new location, but instead show the link to where the agent has migrated. This guarantees that no updates are needed in this node when the agent migrates further. If an agent moves through a node to its original position, the corresponding rules are deleted. To keep tables smaller, we can use a system of wildcards to annotate groups and clusters of agents.

For example, if we migrate `prog.vub.ac.be/WebDispatcher` to the system belnet.ac.be[2], as shown in Figure 1, we take the agent's state and send it as an agent message to the local node `prog.vub.ac.be`. This node sends the agent to its uplink, `vub.ac.be`, and updates its routing table to let `prog.vub.ac.be/WebDispatcher` point towards the uplink. Then `vub.ac.be` sends the agent to `ac.be` and keeps a rule for agent `prog.vub.ac.be/WebDispatcher` which points to its uplink. The node `ac.be` sends the agent to the right downlink, in this case `belnet.ac.be` and makes a rule to point out the agent's new location. If an agent migrates while messages are being sent to it, these messages follow the agent on its path and arrive at the right location.

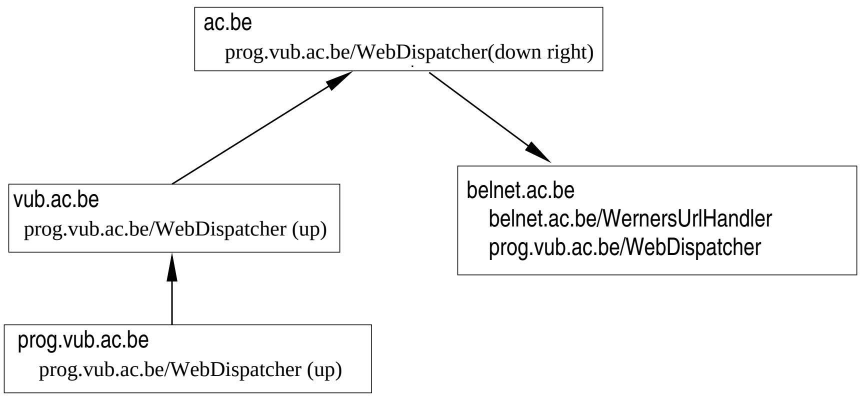

Figure 1. Migrating an agent over the network

[2] We have to admit that a complete migration of the WebDispatcher agent is rather unlikely, instead we expect to see a migration of the UrlHandler Agents. Nevertheless the above is possible and still useful for demonstrative purposes.

3.2: Strong Mobility

Having explained the possibility of sending messages between migrating agents, we now discuss migration in detail. The term migration denotes the act of transferring a running agent to another location. After migration, the agent should proceed seamlessly with what it was doing before it was moved. In our example, we want to migrate the WebDispatcher agent. In order to obtain this result, three actions should be undertaken to prepare, guide and complete the actual migration. First we have to encapsulate the agent's complete state; next, we need to transfer this capsule and, finally we need to restore and re-activate the agent in its new environment. The challenging aspect in migration was the wrapping and unwrapping of the agent in order to restore it to its full powers.

As we said above, we do not want to take mobility into account while programming the WebDispatcher and UrlHandler agents. Therefore, we should be able to migrate the WebDispatcher in the midst of its execution, while it has a runtime stack which is growing or shrinking. In our experience, implementing strong migration was not too troublesome, thanks to the access to a solidly written interpreter. The interpreter (originally called Pico) is written in a thunk-based way (meaning that every evaluation step is stored in the working memory of the running program) and has the capability of serializing the data store. We used this to implement strong migration.

Even if an agent is in the midst of being evaluated, we can simply interrupt the evaluation process whenever the stack is consistent, i.e. after the current continuation thunk has stopped executing, and when all global variables are saved in the data store. We can easily serialize the entire state of the interrupted agent, which also includes the computational state, and send it to another location, after removing the process from interpreter control. At the receiving end we deserialize the agent and start a process which uses the freshly deserialized computational state.

Serializing agents consists of traversing the data graph and storing everything we encounter, including local objects. The one exception is the link with the operating system, which is simply marked as being the root environment. In this way we can integrate agents into their new agent environment upon their arrival at a new location.

3.3: Synchronizing Agents

Now we can easily write WebDispatcher and UrlHandlers. We can name them and they can communicate and migrate freely. But there are still some problems left, such as synchronization between two agents. To use our example: how could we make the WebDispatcher wait for an answer from one specific UrlHandler? If we want to be able to synchronize processes in these kinds of systems we need an underlying fundamental communication model which includes synchronization between processes. We can think of CSP [5] , pi-calculus [7], actor systems [1]. But it turns out that all these models are only valid in small-scale systems because they either use shared memory with locks on variables, or the granularity is too coarse. When using agents, we need a synchronization mechanism which allows for synchronization between processes of different owners, for synchronization while using a network with unpredictable delays, and for data communication between the synchronizing processes.

We found out that a sync-primitive and first class continuations were all we needed to synchronize between multiple agents. The sync-primitive behaves as follows:

Bob	Alice
`sync(''Alice'',[a,10])`	...
Waiting for Sync	...
...	`sync(''Bob'',[20,b])`
Execution continues with a=20	Execution continues with b=10

The arguments of the sync are as follows: the first argument specifies the agent with which we wish to sync. This can be done by explicitly naming that agent, by using a wildcard, or by providing a table containing all possible synchronees. The second, optional, argument is a pattern, which places restrictions on which agents may synchronize. This way we can choose our synchronee by other means than solely its name.

Patterns are unified if and only if they match and there are no free variables after the match. When unification occurs, the agents in question synchronize. The pattern given to the sync operation can contain literals (numbers, strings, booleans, void), variables and tables. Numbers match when they are numerically equal. Strings match when they are equal using a strcmp. Bound variables match if their values match. Free variables match any expression (such as 12, 15, "johan", [10,20,30] and others) and are bound afterwards. Tables match if all the sub-expressions match. For example the table [10,20] matches with [10,20] and [10,a] but not with [20,20]. Finally there is a wildcard called 'any' which matches with anything and forgets the result.

The use of patterns and unification is clear:

- We immediately have two-way communication between synchronizing agents. In the example, when Alice continues after the sync, Alice knows that Bob continues with a=20.
- It allows for a certain selection of who can synchronize. We do not necessarily need to restrict synchronization between *two* agents only. For example, we can synchronize the WebDispatcher with *all* its synchronees in *one* statement.
- Unification is a more dynamic guarding system than CSP; CSP even has more problems because we need roll-backs. Unification does not need this.

3.4: Agent Communication

CBorg agents can send messages to each other using a fairly obvious mechanism. Inspired by the ease of use of RPC implementations, we refer to a remote agent using its name, and sending a message to an agent is similar to sending a message to an object. For example:

```
a:agent("belnet.ac.be/WernersUrlHandler")
                           // creates a reference to the agent WernersUrlHandler
a.showHtml("~werner/index.html")
                           // by sending this message the agent will display
                           // page ~werner/index.html on the local local screen.
```

Sending messages in this fashion allows for asynchronous delivery of a message from one agent to another. The message send command is asynchronous, execution of this instruction solely places the message in the message delivery system. The call immediately returns the value 'void' and the agent continues its execution. This implies that the sender does not wait for the message to actually arrive, which speeds up program execution through bypassing the large delays that can be possible in wide-area networks.

All parameters passed to a remote procedure are automatically serialized. In this process all local objects remain local and are passed as a reference. The main motivation for passing objects by reference is that it allows for easy implementation of callback functions and objects, as is illustrated in the next small example. WernersUrlHandler is the receiving agent, which performs a callback to the second agent, the WebDispatcher. After creating a callback procedure (Display), WebDispatcher calls WernersUrlHandler. The result of the computation is sent back as a parameter of the Display procedure.

```
                          // in prog.vub.ac.be/WebDispatcher
Display(text):: [...]     // Displays the given text
GenerateHtml(Url)::
  { handler:findUrlHandler();
                          // find the agent who will handle this URL.
    handler.generateHtml(Url, agentself())
                          // ask the agent to generate HTML and send
                          // it back to myself. (hence the agentself parameter)
  }
                          // in belnet.ac.be/WernersUrlHandler
generateHtml(Url, Callback)::
  Callback.Display("<HTML>Online and kicking !</HTML>");
```

4: Writing Mobile Agents

This section covers our experience in writing mobile agents. This experience has been gained by guiding numerous students in writing agents and also by implementing some larger applications, like our web server example, written in an ‘agent-oriented fashion’.

When working in a distributed architecture that supports code mobility, we need to write applications in a new fashion. This section explains some of the design rules we used when writing mobile agents. These rules should be viewed as a guide, based upon our experience and which are suited to our needs. They are, however, not the one and only approach to writing solid mobile code.

4.1: Asynchrony

> Try to do as much as possible in an asynchronous fashion: send messages to other agents and do not wait for the answers. Instead, just specify where the answer has to go to.

The most pressing issue in mobile computing is that we are working in a distributed environment. This means that we have various delays for messages between agents. An agent could send a message to another agent, and wait for an answer before continuing. However, this delays its own execution speed due to the slow message delivery speed of the network.

For example, we could write the WebDispatching agent as shown in Figure 2. If we take the total time needed by the WebDispatcher to handle two different requests, we see that the total time is the sum of handling both requests, which is of course too slow if we have to handle a large number of requests at the same time.

Another possibility (illustrated in Figure 3) is that an agent can start handling other requests while waiting for an answer by relying on a callback from the called agent. This approach has been shown to execute faster and to be more reliable, but it requires a new, non-standard way of thinking (especially for people who are not used to working in distributed environments). Figure 3 shows that the total time needed to handle both requests is usually the maximum time needed by one of both UrlHandlers, which is better than the first implementation.

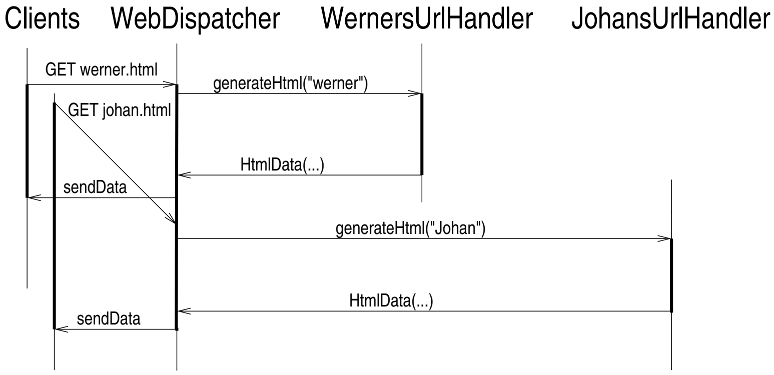

Figure 2. Sequenced implementation of the WebDispatcher

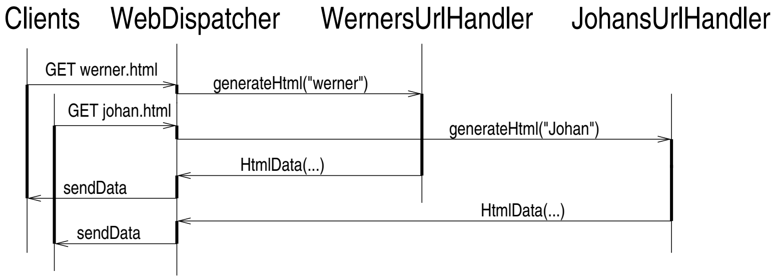

Figure 3. Asynchroneous implementation of the WebDispatcher

Working asychronously also means that we should never write an agent that 'returns' a result (return in the sense of a return statement in object oriented languages and imperative languages). To continue the computation, an agent sends a message to another agent. In practice this means that almost every incoming message should have a 'send-the-result-to' field, that specifies what to do with the result. The caller can either choose to send the answer back to itself to continue the computation, or it can choose to send the message to another agent, which handles the rest of the computation.

We have implemented this in our system, using an arrow syntax, which automatically adds a 'result- to' argument to messages. The arrow operator defines where the result of a computation has to go when completed and a result is available. For example:

```
agent2.Calculate(<something>) -> agentself
```

This specifies that the result of Calculate should be sent back to the sending agent.

4.2: Autonomy

> An agent should behave autonomously, meaning that its correct local execution should not depend on its communication partners.

When writing agents in a distributed system, we often encounter the problem of partial failures. For example, whenever one of the communication partners of a given agent dies, the agent will stop working correctly because it is waiting for some action of the dead partner. It is clear that this should not be allowed. Sadly, partial failures are everyday facts in distributed systems, and cannot be easily abstracted nor can they be ignored. So when designing agents, failure of external factors should be taken into account and a good error-resolution strategy should be conceived.

For example, the WebDispatcher's design is such that an incoming call is redirected to the specific UrlHandler. If the UrlHandler crashes or does not respond, nothing happens, which is not a problem, because the WebDispatcher agent will continue working.

Being autonomous also means having a complete execution space in which to work, without interference from others. All the elements under an agent's supervision should be owned by this agent, and only by this agent. This also explains why agents should have their own code and data spaces, and an independent thread of execution.

4.3: Modularity

> Agents mind their own business and are loosely coupled. They make no assumptions about the overall computation being performed.

The concept of being autonomous can be extended to the fact that an agent should not place certain requirements upon the calling sequence followed by a sent message. Neither can we assume that whenever a message arrives this message has followed a certain path. We receive a message and we handle it, without knowledge of the overall computation. We should never think of an agent as working in cooperation with other agents. We think of a agent as offering a certain service that can be used by others, and we don't care about what the others want to do with this service. This is because we are working in a dynamic environment, where we cannot foresee every possible use of an agent. This also gives rise to modular, loosely coupled software, which is exactly what we need in distributed mobile systems.

In the WebDispatcher agent this is noticeable in the fact that we simply ask the UrlHandlerAgent to call us back with a certain result. How this UrlHandler obtains its result does not matter. If a certain UrlHandler needs to subscribe to a certain URL to actually generate the html, we do not care, we regard this 'Subscribe' message as a completely new message. In the Message Sequence Chart shown in Figure 6 this can be seen in the gray box. The gray box is the subscribe request to subscribe to a certain URL. This gray box behaves exactly the same whether it was initiated as a result from a previous message send or not. This agent would not behave modularly when the Subscribe message behavior would depend on the context it was used in.

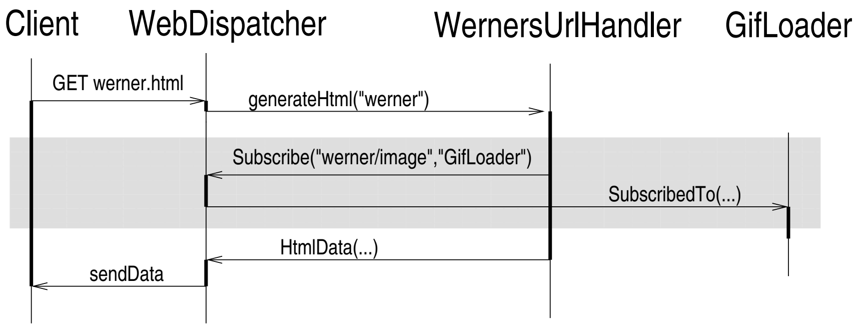

Figure 4. Message Sequence Chart sketching modularity

4.4: State-Based

> Try to write agents in a state-based fashion.

One of the easiest ways to make agents autonomous and asynchronous is by using a state-based approach. We suggest using a kind of Finite State Machine description, which describes the agents' behavior.

However, a significant problem with implementing state-based approaches is the lack of 'history'. For example, it is very difficult to write a state-based machine that resumes its calculation when some answer arrives from an external agent, because we have to restore a certain context before we can continue with the computation. A standard solution to this is using multiple threads, each handling a different calling sequence. But this usually leads to concurrency problems[3].

A better solution we are using now is a restricted form of concurrency: we think of the agent as an active, single-threaded entity, which switches between states. Of course, using this quite straightforward mental image in contemporary languages is not easy, because saving and restoring states is troublesome. Therefore the language in which a mobile agent is written should, ideally, support first-class continuations. We have implemented first-class continuations in CBorg, using the concept of 'return' continuations.

5: Conclusion

This paper describes some of our experiences in mobile computing. We have shown that the current infrastructures do not provide adequate support for mobile components. Therefore, we implemented the CBorg mobile agent infrastructure, to allow us to experiment with mobile components. Some of the more notable features of CBorg which aid in implementing mobile components are a location-transparent distribution layer, strong mobility, the ability to synchonize agents, and easy agent communication.

Our experiences with CBorg have shown that mobile computing requires a different way of thinking when designing and writing software. We stated a number of design rules which aid in writing these components: Asynchrony, Autonomy, Modularity, and using State-based components.

References

[1] G. Agha. Actors: A model of concurrent computation in distributed systems. Technical Report AI Tech Report 844, Massachusetts Institute of Technology, 1985.

[2] A. Chavez, D. Dreilinger, R. Guttman, and P. Maes. A real-life experiment in creating an agent marketplace. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, April 1997.

[3] Theo D'Hondt. http://pico.vub.ac.be.

[4] R.S. Gray. Agent tcl: A transportable agent system. In J. Mayfield and T. Finnin, editors, *CIKM*, 1995. Workshop on Intelligent Information Agents.

[5] C.A.R Hoare. Communicating sequential processes. *Prentice Hall International Series in Computer Science*, 1985.

[6] Norman C. Hutchinson. Emerald: An object-based language for distributed programming, January 1987. Department of computer science, University of Washington.

[7] Robin Milner. The polyadic pi-calculus: A tutorial. LFCS report ECS-LFCS-91-180.

[8] Charles E. Perkins. Ip mobility support. Sun Microsystems, October 1996.

[9] Gian Pietro Picco. Understanding code mobility. Tutorial at ECOOP98, July 1998. Politecnico di Torino, Italy.

[10] K. Rothermel and R. Popescu-Zeletin. Mobile agents. *Lecture Notes in Computer Science*, 1219, 1997.

[11] Werner Van Belle. http://prog.vub.ac.be/CBorg/.

[12] Werner Van Belle. Reinforcement learning as a routing technique for mobile multi-agent systems, April 1997.

[13] Werner Van Belle. Location transparent routing in mobile multi-agent systems: Merging name lookups and routing. In *Future Trends In Distributed Computing*, 1999.

[3] UML and OMT currently allow for history states, nevertheless it is important to notice that this history state does not solve any problems as long as we do not combine it with some form of concurrency management.