Distributed computing and code mobility

The purpose of abstraction is to facilitate research and provide noticeable progress in a particular endeavor. Classic examples are modern algebraic structures like Groups, Rings etc. Classic computing examples are Abstract Data Types, Algorithm Efficiency etc.

In mathematics, among other things, we are interested in answering questions like the solvability of equations by radicals. In computing, among other things, we are interested in automating certain aspects of computation, such as the statements of structured programming.

This article presents an abstraction for automating operations of distributed computing and code mobility. The abstraction illustrated in this article has two ends. One end is at the distributed computing itself, and the other end is at operating system. Indeed, the infrastructure can only be in the form of a distributed operating system. Thus, the article introduces an abstraction, as well as an infrastructure that supports the abstraction.

The formalism presented for the mechanization of distributed computing is general because it is free of impurity. The linguistic statements do not resemble or involve any form of hardware or other technology dependency that may change over time or place.

Distributed computing

Older languages reflected hardware capabilities, such as FORTRAN three-way selector. Eventually, structured programming formalized the notions of selections and iterations. Accordingly, we need to separate the design of distributed algorithms from the interaction and mobility of the components that contain those algorithms. Indeed, algorithmic abstractions are generally conveyed in pseudo-code, which get buried under implementation details pertaining to communication of components and code mobility.

From the operational point of view, we deal with locations of the components, their intercommunication and mobility. The parts of an application that deal with component interactions should not involve any code with the knowledge of the location of those components.

An application and its components should be able to move from node to node. In a distributed algorithm, an object (component) may move from one node to another maintaining its state as it travels. It may then report its findings at some point into the algorithm that it carries along.

Operating system

An operating system can be viewed from different perspectives. The standard view depicts an operating system as a sophisticated device driver, interacting with sound, graphic, memory and storage controllers on behalf of applications.
An operating system also manages abstractions like threads, processes and their interactions. Furthermore, some devices such as file storage, keyboard and screen, can be handled through abstractions like the notion of streams. **In order to achieve a distributed operating system, we need to separate and classify its functions until we arrive at an abstraction that is essential for distributed computing.**

The fundamental element in our context is the notion of a component. What each component does is internal to that component. We are interested in a component as a multi-threaded process. A **distributed application comprises of communicating components executing on various nodes of a heterogeneous network.**

An abstract distributed operating system that deals with processes, threads and their intercommunication will be a leap in the implementation of distributed algorithms. It will free us from having to deal with multiple languages and libraries for what is not relevant to the actual intent of the application.

**Formalization of abstraction**

Software is written in a language. Therefore, we start with examining the language that will mechanize the operational semantics of distributed computing.

A component should not be distinguishable from a standalone application. The distinction will result in having to remember different encoding and maintenance procedures. Furthermore, without such distinction one can compose a new application that will interact with already **executing** applications. The new application will simply view the existing applications as some of its components. **Some uses of this idea are known as component-oriented development.**

Now, we have to solve an interesting problem. For an application to interact with a remote component, it must know where to look, and what type of messages to send. However, we do not want the statements of the new application to be sprinkled with knowledge about the specifics of the component being referenced. We solve this problem in two steps. Keep in mind that component and application mean the same thing.

The boundary of a component is the set of messages that it will accept. A boundary message is expressed as a global function specified as “entry”, like the following.

```c
entry int perform(double);
```

A component will usually provide several entry points as doors at its boundary. For purposes of illustration, one message is sufficient. The boundary of a component is expressed using the class type constructor. We make a header file, as shown below, for **users** of this component.
class RemoteComponent = URL
public:
  external int perform(double);
end;

The URL is known when we create the remote component. The users of the component need to include the above class and derive from it in any way they wish. The important thing here is that the compiler has all the information that it needs in order to generate code for calls to the method “perform”. Furthermore, the compiler knows that it must generate code for the body of the method because it has been specified as “external”.

**Note that there will be no reference in the statements of the application that will include any knowledge of the remote component.** The application will simply derive from the class “RemoteComponent”, and define its own methods. The derived class can also add constraints and invariants, as it may need. Furthermore, the application can directly call the method “perform”, even though it never defined a body for the method.

The compiler uses the URL when it elaborates a declaration for an instance of the class “RemoteComponent”. However, the statements of the application using this class are free of pollution.

**Code mobility**

Now that we have presented the linguistic constructs, we need to build an infrastructure, a compiler and so on. In other words, we need a distributed operating system that supports mechanisms needed by our linguistic abstractions, and a compiler to map the abstractions to those mechanisms. **Actually, that part has already been done.** The fact is that an application using a remote component, such as “RemoteComponent” of previous section, can freely move from node to node, as illustrated in next section.

Code mobility refers to an executable (binary) moving from one node to another. In order to interact with a remote component we only need the URL leading to the component. Therefore, the executable code of an application using the remote component can reside on any node. Hence, the application can simply travel while keeping its contact with the remote component.

**Autonomous or Mobile agents**

Now, let us reverse our view of the above scenario. Let the remote component be the central application, and the application using the component be an agent. In this scenario, the agent can move anywhere while maintaining its contact with the central application.

Presumably, the messages sent by the agent to the application, determine its next course of action, or destination. Indeed, in an enterprise an agent can simply determine its next destination from a table lookup and its state.
A mobile agent may have access to many enterprise components like our example of “RemoteComponent”. Some of these components may interact with corporate databases, security modules or anything else for that matter.

An agent moves itself as shown below.

```
travel destination_address;
```

The “travel” statement needs the IP address of the node to reach. However, the agent determines the address during execution.

The semantics of “travel” are quite intuitive. The agent leaves the current node and begins execution at destination node, starting with the statement following the travel statement. **Thus, the agent’s execution terminates at current node and continues at destination node.** This type of mobility is often referred to as “strong mobility”.

Whether or not the algorithm of a mobile agent is intelligent is not related to its ability to move and communicate with remote components. However, without an abstraction for code mobility and an infrastructure to support the abstraction, a substantial amount of development time is spent in dealing with such irrelevant matters.

**Conclusion**

The language we used for illustration is Z++. The distributed operating system we mentioned is the Z47 Processor. **Thus, one can implement distributed algorithms in Z++ without spending time on the physical distribution of parts of the algorithm and their communication, or any platform considerations for that matter.**

The distributed operating system Z47 provides the infrastructure for code mobility and autonomous agents. Furthermore, the Z++ formalism provides direct and intuitive linguistic support for component-oriented development.