Towards a Practical π-Calculus Based Mobile Agent System

Seiji Umatani, Masahiro Yasugi, and Taiichi Yuasa

Graduate School of Informatics, Kyoto University

In recent years, almost all computers around the world (such as servers, PCs, and PDAs) are connected to the Internet and many people want to utilize remote resources effectively via the network. In such situations, *mobile agent* model gets a lot of attention as one of powerful computation models which enables the construction of flexible, scalable, and secure distributed applications in a wide-area computer network.

We are now designing and implementing a π-calculus based mobile agent programming language. *π-calculus* is a concurrent process model in which multiple processes interact with each other via channels. There are many research activities which propose various extensions to π-calculus with the concepts of *location* and *mobility*. However, most of them are developed for the purpose of theoretical study, and there are not many practical π-calculus based systems at present.

π-calculus has several problems to be solved in order to be used as a practical programming language. The most serious one is the runtime resolution of names. In π-calculus (and many its extensions), the reduction of a process term is defined by a set of reduction rules and each name occurrence in the term can be resolved by searching for the corresponding channel declaration through enclosing terms at runtime. In order to pass a locally declared channel to another process, the scope of the channel can be arbitrarily extruded to include the process that receives it. This extrusion, however, is possible only in the impractical assumption that the entire system is described in a single source program. From a practical point of view, realizing scope extrusion needs some kind of elaborate means to avoid name collisions, to refer to names declared in other modules, and so on.

On the other hand, we employ π-calculus as a base model of our language because its simple syntax and semantics make program analysis simple and straightforward. In implementing a practical system, we need to consider the execution efficiency of mobile agent programs, and if their analyses are simple and easy tasks, we can apply various optimization techniques to them. In contrast, with mobility extensions to an existing language (e.g., Java), it takes a considerable effort for these extensions to be consistent with pre-existing features of the language and it is difficult to analyze correctly.

Based on the above consideration, we designed a new mobile agent language with the syntax shown in Figure 1. In our language, an *agent* is represented as a logical location (written as "`l[...]`") which may include channels, input/output processes, and other agents in it. The main features of our language are as follows:

```
<proc> ::=
      <name> ! ( <expr> [, <expr>]* )
    | <name> ? ( <identifier> [, <identifier>]* ) -> <proc>
    | ( <proc> [| <proc>]* )
    | new <identifier> : [chan | loc] in <proc>
    | <identifier> [ <proc> ]
    | move <identifier> then <proc>

<expr> ::= ... | <name> | ...
<name> ::= <identifier> | '<identifier>'
```

Figure 1: The syntax of our mobile agent language (excerpt).

flexible name resolution: Normal name occurrences in a program are resolved with lexical scope at compile time. During its execution, each name keeps referring to the same channel bound at compile time independently of the enclosing agent's movement. At the same time, runtime name resolution with dynamic scope may be preferable in some situations. For such situations, our language provides another way of specifying channels with dynamic scope (denoted with quotation, like `'c'`).

first class location: In our language, locations as well as channels can be passed among agents through channels. This enables the construction of flexible and modularized large-scale applications because we can almost always keep locations separate (i.e., invisible) from each other and we may pass them through channels.

strict definition of channel affiliation: During the execution, a channel is created at the *current* location when its declaration is executed and it never changes its affiliation once created; that is, if a location (agent) moves to another location, it takes its affiliate channels with it. This semantics simplifies the implementation and makes it easier for programmers to figure out communication patterns. In theoretical terms, channel affiliation means that no scope extrusion is permitted in our language.

In our current implementation, the entire system consists of location servers and consoles, both written in Java. A location server corresponds to a single (logical) location, and all servers are interconnected with each other in a tree-structure manner. A server executes arriving agents in turn with a naive interpreter. A console can connect to any location server and it can send an agent program to the server interactively.

Additionally, in the current implementation in Java, we can register any Java object as a passive and immovable agent in a server. In order to call this agent's Java method, another agent at the server can send an appropriate message to it through a channel that can be resolved only with dynamic scope.