Abstract Machines for Safe Ambients in Wide-Area and Mobile Networks

Seiji Umatani, Masahiro Yasugi, and Taiichi Yuasa

Graduate School of Informatics, Kyoto University,
Sakyo-ku Kyoto 606-8501, Japan
{umatani,yasugi,yuasa}@kuis.kyoto-u.ac.jp

Abstract. Recently, there have been several studies focusing on the implementation of process calculi with distribution and mobility. Among these, PAN and GCPAN are distributed abstract machines for executing Safe Ambients, a variant of the Ambient calculus. However, in order to use them or to exploit their implementation techniques, we must assume all-to-all and permanent connectivity in the underlying network; this is inappropriate for most real-world wide-area and mobile networks, in which each private network is delimited by network boundaries and each mobile device may become disconnected at any moment. In this paper, we propose novel abstract machines PAN_{mov}, $\text{GCPAN}_{\text{mov}}$, and $\text{GCPAN}_{\text{shift}}$ that can handle such network boundaries and mobile devices by using a special kind of agents called *boundary forwarders*. Especially in $\text{GCPAN}_{\text{shift}}$, operations related to boundary forwarders improve the fault tolerance of user programs. Finally, we prove the correctness of the proposed machines by using weak barbed bisimulation.

1 Introduction

In recent years, core calculi based on distribution and mobility have been studied extensively, and they are regarded as fundamental models for many programming languages supporting code migration.

The Ambient Calculus (AC) [2] is a distributed process calculus with a notion of locations that are known as *ambients*. Each process belongs to an ambient, and each ambient, except for the topmost ambient, belongs to another ambient. Thus, the ambients form a hierarchical structure, and every process belongs somewhere in the hierarchy. In AC, computation is represented as the combination of three types of primitive operations of ambients—`in`, which instructs an ambient to enter another ambient; `out`, which instructs an ambient to exit from another ambient; and `open`, which provides a way to dissolve the membrane of an ambient so that the content of the ambient can be accessed.

To realize practical programming language systems adaptable to a broad range of heterogeneous distributed and mobile networks, we can adopt AC as the core language of such systems. Several studies have been carried out on the distributed implementation of Ambient-like calculi [1, 4, 12, 8, 11, 13]. Among

these, PAN [12, 7] and GCPAN [8, 9] are abstract machines for implementing the Safe Ambients (SA), a variant of AC, in distributed settings.

The implementation technique of PAN and GCPAN is simple; it separates the *logical* distribution of ambients, which is given by the hierarchical structure of ambients, from their *physical* distribution, which is a mapping from each ambient to a certain computer on which the ambient is running. PAN and GCPAN exploit this separation to defer physical movements of an ambient that executes some `in` or `out` moves, until the ambient is opened by the target ambient so that it can access the contents of the target ambient, such as files. This implementation technique may eliminate unnecessary communications caused by physical movements, which are typically much more expensive than simple data communications. In particular, when an ambient moves to its target ambient through multiple moves and accesses the contents of the target, physical movements corresponding to intermediate moves could be eliminated entirely. Moreover, if the moving ambient does not access any content of the target, there is no need for physical code migration.

This technique of PAN and GCPAN requires all-to-all and permanent connectivity of the underlying network; that is, it assumes that any computer in the network can directly communicate with the other computers at any moment. In particular, the computer to which an ambient has moved through (possibly) multiple `in` and `out` moves must be able to communicate with the computer in which the ambient physically resides when the ambient is opened. An ambient may move to any target ambient; hence, any computer can become the target location. Therefore, any computer must be able to communicate with the other computers.

Clearly, the above requirement is not desirable for wide-area distributed environments which consist of several local-area networks or for mobile networks in which mobile devices such as smartphones may be disconnected temporarily or permanently at any moment. Furthermore, deferring the movement of an ambient is not desirable if we want the ambient to move only for using the CPU power of remote computers.

In this paper, we propose novel abstract machines PAN_{mov}, $\text{GCPAN}_{\text{mov}}$, and $\text{GCPAN}_{\text{shift}}$ for SA in order to support such wide-area and mobile networks. The main ideas of the proposed technique are as follows:

1. We model a wide-area network as a set of network domains, each of which is isolated from other network domains by network boundaries. Computers within a particular network domain can directly communicate with each other. We extend SA's ambient creation construct $M[P]$, where M is the name of a created ambient and P is its content, to make it clear whether the created ambient belongs to the same network domain as its parent or not; if not, there exists a network boundary between them.
2. We modify the implementation of `in` and `out` moves so that an ambient physically moves to the target ambient as soon as the ambient performs a cross-boundary movement. When physical movement of an ambient is performed in PAN (and also in our machines), a special agent called *forwarder* is

created at the original location of the ambient; after that, all messages sent to the ambient from its children are transferred via this forwarder. Thus, our technique guarantees that the content of an opened ambient can always be sent to its parent through a chain of forwarders even if the ambient and its parent belong to different network domains.

The proposed technique is based on the PAN abstract machine. In particular, movements within a single network domain are processed in the same way as they are in PAN. Thus, no physical code migration is performed unless some `open` action or some cross-boundary movement is executed.

In the proposed abstract machine, every cross-boundary movement causes physical code migration; hence, it requires the creation of more forwarders than PAN, wherein all movements are considered as non cross-boundary movements. Although such an increase of forwarders is inevitable in wide-area networks with boundaries, keeping unused forwarders (i.e., forwarders with no child) alive is a waste of computer resources. Therefore, we have refined the proposed abstract machine so that it can collect and reuse the resources assigned for unused forwarders in the same way as GCPAN. Furthermore, to reduce the number of forwarders in use and to prevent erratic behavior related to failures in network communication, the proposed abstract machine relocates ambients that use forwarders so that they can directly send messages to their parents.

We provide a formal description of the proposed machine and we prove its correctness by establishing a bisimilarity between it and PAN. In the formal description, the underlying protocols for implementing ambient (co)capabilities follow the formalization method of PAN. In terms of the correctness of the proposed machine and reusability of the real implementations of PAN (and GCPAN), the fact that the proposed machine is a smooth extension of PAN is advantageous.

Cross-boundary communication and code migration are common features of recent distributed systems; hence, we believe that the proposed ideas and techniques can be adapted to such systems.

The remainder of this paper is organized as follows. In Section 2, we provide a brief explanation of SA and PAN. Next, in Section 3, we describe the proposed base abstract machine PAN_{mov}, which does not perform garbage collection. In Section 4, we explain how one of the refined abstract machines, $\text{GCPAN}_{\text{mov}}$, reduces the resource usage of the underlying computers. In Section 5, we explain how another refined machine, $\text{GCPAN}_{\text{shift}}$, further reduces the resource usage and improves the fault tolerance of user programs. In Section 6, we prove the correctness of PAN_{mov}. Finally, in Section 7, we provide concluding remarks.

2 Background

2.1 Safe Ambients

Safe Ambients (SA) are ambients that are designed to prevent unintended interference among ambients; whenever an ambient executes some movement action,

the target ambient of the movement must execute the corresponding *coaction*. Thus, the timing of each movement can be controlled by the target ambient, and no unintended interference occurs.

The kinds of processes in SA are the same as those in the original Ambient calculus [2]: $P_1 \mid P_2$ for parallel composition, $(\boldsymbol{\nu} n)P$ for restriction, $M.P$ for action prefix, $M[P]$ for ambient creation, $\langle M \rangle$ and $(x)P$ for local communication, and X and $\texttt{rec}\ X.P$ for recursive processes. The characteristic of SA is found in the existence of coactions in M:

$$M ::= x \mid n \mid \texttt{in}\ M \mid \overline{\texttt{in}}\ M \mid \texttt{out}\ M \mid \overline{\texttt{out}}\ M \mid \texttt{open}\ M \mid \overline{\texttt{open}}\ M$$

where each action (`in`, `out`, and `open`) must be executed along with the corresponding coaction ($\overline{\texttt{in}}$, $\overline{\texttt{out}}$, and $\overline{\texttt{open}}$, respectively). The following are the reduction rules that define the behavior of basic actions in SA processes:

$$\begin{array}{ll}
\text{[R-Msg]} & \langle M \rangle \mid (x)P \longrightarrow P\{M/x\} \\
\text{[R-In]} & m[\texttt{in}\ n.P_1 \mid P_2] \mid n[\overline{\texttt{in}}\ n.Q_1 \mid Q_2] \longrightarrow n[\,m[P_1 \mid P_2] \mid Q_1 \mid Q_2] \\
\text{[R-Out]} & m[\,n[\texttt{out}\ m.P_1 \mid P_2] \mid\ \overline{\texttt{out}}\ m.Q_1 \mid Q_2] \longrightarrow n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2] \\
\text{[R-Open]} & \texttt{open}\ n.P \mid n[\overline{\texttt{open}}\ n.Q_1 \mid Q_2] \longrightarrow P \mid Q_1 \mid Q_2
\end{array}$$

2.2 Pan Abstract Machine

To express how SA can be executed in a network of computers, the Pan abstract machine consists of a flat network of *located agents*. Intuitively, a located agent $h : n[P]_k$ represents an ambient n whose content process is P, where h is the physical location of n and k is the physical location of its parent ambient. Thus, for example, the logical hierarchical structure of the SA process $\texttt{a}[\texttt{b}[P|\texttt{c}[Q]]|R]$ is represented by the parallel composition $h_1 : \texttt{a}[R]_{\texttt{root}} \parallel h_2 : \texttt{b}[P]_{h_1} \parallel h_3 : \texttt{c}[Q]_{h_2}$ in Pan. Note that ambients in Pan do not need to know the locations of their children. This is because the existence of coactions in SA guarantees that an interaction among ambients is always triggered by a child ambient using an upward request message to its parent, as described below. The precise syntax of Pan is a subset of that of Pan_{mov}, which is given in Section 3.2.

The basic actions of SA are simulated in Pan using interactions among ambients, as shown in Figure 1. In the figure, the white boxes represent located agents, and the lines between them represent parent-child relations. The arrows denote messages between them. In Figure 1 (b), for instance, the ambient `a`, which performs $\texttt{out}\ \texttt{b}.P$, sends a request message $\{\texttt{out}\}$ to `b`. If `b` contains an unguarded process $\overline{\texttt{out}}\ \texttt{b}.Q$, these action/coaction match in `b`, and `b` sends back to `a` a completion message $\{\texttt{go c}\}$. When `a` receives the completion message, it updates its parent location with the location of `c`. The `in` action is simulated in a similar manner, as shown in Figure 1 (a). It should be noted that these simulations do not change the physical location of `a`; they update only the local information about the parent location of `a`, whereas `a` remains at the same location.

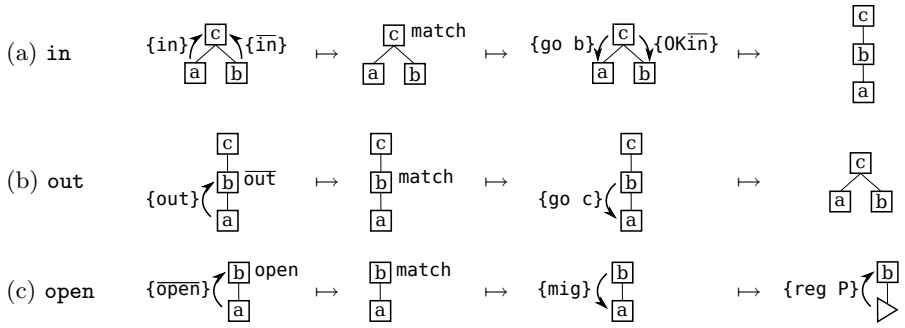

Fig. 1. Simulation of SA actions in PAN.

On the other hand, in Figure 1 (c), when the ambient `a`, which performs a $\overline{\texttt{open}}$ coaction, receives the corresponding completion message {`migrate`}, it further sends back to `b` the completion message {`register` P}, which registers P, the code of `a`'s local processes, into `b`. This incurs the physical migration of P. After sending the {`register` P} message, the ambient `a` becomes a forwarder (depicted as a triangle in the figure), whose role is to transfer messages from its children to `b`. Such a forwarder is necessary because `b` cannot access its children; therefore it cannot inform them to send their requests to `b` instead of `a`. In the remainder of this paper, we use the textual notation $\triangleright\{P\}$ to denote a forwarder. For instance, the final state of Figure 1 (c) is represented as $\texttt{b}[\,\triangleright\{\ldots\}\,]$.

By deferring physical code migration until the containing ambient is opened, these simulations eliminate many unnecessary network messages.

However, the simulations described above have a serious disadvantage in wide-area distributed environments with network boundaries. A set of `in` and `out` moves of an ambient running on a certain computer may *logically* move into another ambient running on a different computer. If these computers belong to different network domains and if they cannot communicate directly, the former ambient can no longer send messages to its parent, i.e., to the latter ambient, whereas PAN assumes that every computer can communicate directly with any other computers. Consider, for instance, the following code representing the firewalls of two LANs as two sibling ambients `f` and `g`:

$$\texttt{r}[\,\texttt{f}[\,\overline{\texttt{out}}\ \texttt{f} \mid \texttt{a}[\,\texttt{out f.in g.}\overline{\texttt{open}}\ \texttt{a} \mid P\,]\,] \mid \texttt{g}[\,\overline{\texttt{in}}\ \texttt{g.open a}\,]\,]$$

where ambients within `f` cannot communicate with `g` directly. After `a` performs `out f.in g`, it is logically placed in `g`, whereas it physically remains within `f`. Then, an attempt to send a message {$\overline{\texttt{open}}$} fails. In such a case, the {$\overline{\texttt{open}}$} and the following {`register` P} messages should be transmitted via `r`, that is, via the path through which `a` moved.

Furthermore, suppose `g` is a mobile device that becomes disconnected before it performs `open a`. Since the code P is not yet delivered to `g`, some intended behavior of `g` in P is lost. (This problem is further discussed in Section 5.)

In summary, PAN's assumption of all-to-all and permanent connectivity among computers is not practical in contemporary wide-area and mobile networks; therefore, an alternative technique is required to express ambient movements.

3 PAN$_{\mathbf{mov}}$: Chaining Forwarders upon Movement

In this section, we propose a novel abstract machine PAN$_{\mathrm{mov}}$, which solves the problem of PAN described in the previous section.

3.1 Basic Idea

PAN$_{\mathrm{mov}}$ solves the problem using two ideas: (1) to specify boundaries between network domains, we slightly extend SA's ambient creation construct, and (2) upon each cross-boundary `in` or `out` move of an ambient, PAN$_{\mathrm{mov}}$ physically moves the ambient into the destination network domain. These ideas are explained in detail below.

Ambient creation with `split` prefix

To handle network boundaries properly, if a new ambient is created in a different network domain from the domain in which the creating ambient resides, PAN$_{\mathrm{mov}}$ places a special kind of forwarder called *boundary forwarder* between them; that is, the parent of the created ambient is the boundary forwarder, and the parent of the boundary forwarder is the creating ambient. The need for a boundary forwarder at each ambient creation could be automatically determined by an actual implementation if the topology of the underlying network is given, for example, as an external configuration file. However, in this paper, to simplify the formal definition of PAN$_{\mathrm{mov}}$, each ambient creation involving the creation of a boundary forwarder is prefixed with the keyword `split` (e.g., `split a`$[P]$). If a programmer writes an SA program with `split` prefixes in accordance with the topology, the program could be executed without any external configuration file. Similarly, a boundary forwarder is placed between each mobile device and its infrastructure.

Besides performing the normal task of forwarding messages, boundary forwarders play several roles in PAN$_{\mathrm{mov}}$, as described below. To distinguish a boundary forwarder from a normal forwarder, we denote the former as $\rhd^{\bullet}$ and the latter as $\rhd^{\circ}$. Sometimes, we simply denote the latter as $\rhd$.

Physical migration upon `in` and `out`

In PAN$_{\mathrm{mov}}$, physical migration of ambients upon cross-boundary movements is achieved with a mechanism that is similar to that of PAN with which it performs `open` actions. The mechanism involves the following steps:

1. If a request message crosses a boundary, the corresponding boundary forwarder marks it with a special tag.
2. When an ambient receives a request message marked with the special tag, it creates an *empty* clone of the requesting ambient at that location.

3. The receiving ambient sends back to the requesting ambient the $\{\texttt{migrate}\}$ message, which indicates code migration into the clone.
4. When the requesting ambient receives the $\{\texttt{migrate}\}$ message, it sends back the $\{\texttt{register}\ P\}$ message containing its content, and then, it becomes a forwarder, as in the case of $\overline{\texttt{open}}$ action.

Following these steps, whenever an ambient moves, a forwarder is created at its original location. Then, even if the ambient repeats several movements, the chain of created forwarders constitutes the path from the original location to the final destination along which all messages sent from its children can always be transmitted.

Note that the physical migration of a moving ambient is performed only when its request message crosses a network boundary. In other words, every movement of an ambient within a single network domain is treated as it is in PAN; hence, physical migration is deferred until it is opened later.

If a request message has crossed a network boundary, the $\{\texttt{migrate}\}$ message of step 3 and the $\{\texttt{register}\ P\}$ message of step 4 must also cross the network boundary. Moreover, at step 4, the requesting ambient cannot send the $\{\texttt{register}\ P\}$ message via the boundary forwarder of step 1, which forwards request messages to the parent, because the destination of the $\{\texttt{register}\ P\}$ message is not the parent, but its own clone. To remedy these difficulties, the boundary forwarder creates, at step 1, a *seed* of a boundary forwarder, denoted by $\bullet$. The $\{\texttt{migrate}\}$ message sent back from the parent ambient is transmitted backward by this seed. Furthermore, after transmitting the $\{\texttt{migrate}\}$ message, the seed becomes a new boundary forwarder targeting the appropriate location. For instance, consider the following code:

$$\texttt{root}[\,\texttt{a}[\,\overline{\texttt{out}}\ \texttt{a}.P \mid\ \rhd^{\bullet}\{\,\texttt{b}[\,\texttt{out}\ \texttt{a}.Q \mid \texttt{c}[\,R\,]\,] \mid S\,\}\,]\,]$$

When $\texttt{b}$'s $\{\texttt{out}\}$ message arrives at $\texttt{a}$ via the boundary forwarder, the state changes to:

$$\texttt{root}[\,\texttt{a}[\,P \mid\ \bullet\ \mid \rhd^{\bullet}\{\,\texttt{b}[\,Q \mid \texttt{c}[\,R\,]\,] \mid S\}\,]\,]$$

Then, when $\texttt{a}$'s $\{\texttt{migrate}\}$ message arrives at $\texttt{b}$ via the seed, the state changes to:

$$\texttt{root}[\,\texttt{a}[\,P\,] \mid \texttt{b}'[\,\rhd^{\bullet}\{\,\texttt{b}[\,Q \mid \texttt{c}[\,R\,]\,]\,\}\,] \mid\ \rhd^{\bullet}\{\,S\,\}\,]\,]$$

where the seed becomes the boundary forwarder targeting $\texttt{b}'$, the clone of $\texttt{b}$. Finally, $\texttt{b}$'s $\{\texttt{register}\}$ message is correctly sent to $\texttt{b}'$ via the new boundary forwarder, and the state becomes:

$$\texttt{root}[\,\texttt{a}[\,P\,] \mid \texttt{b}'[\,Q \mid\ \rhd^{\bullet}\{\,\rhd^{\circ}\{\,\texttt{c}[\,R\,]\,\}\,\}\,] \mid\ \rhd^{\bullet}\{\,S\,\}\,]\,]$$

Note that this new boundary forwarder continues to work for its children after these transitions; in the code stated above, all messages from $\texttt{c}$ are forwarded by it.

In the mechanism described above, there is another subtle difficulty at step 2. If a request message is $\{\mathtt{out}\}$, or if it is $\{\overline{\mathtt{in}}\}$ without the special tag (i.e., the message did not cross any boundaries), a clone may be allocated at the same domain as the receiving ambient. On the other hand, if the message is $\{\overline{\mathtt{in}}\}$ marked with the special tag, a clone must be allocated at the same domain as the ambient that sends the $\{\overline{\mathtt{in}}\}$ message. For instance, in the code:

$$\mathtt{root}[\,\mathtt{a}[\,\mathtt{in\ c}.P \mid \mathtt{b}[\,Q\,]\,] \mid\ \rhd^{\bullet}\{\,\mathtt{c}[\,\overline{\mathtt{in}}\ \mathtt{c}.R\,]\,\}\,]$$

the clone of `a` must be created below the boundary forwarder. Here, `c` cannot predict whether its $\{\overline{\mathtt{in}}\}$ message crosses any boundary forwarder when it emits the message; hence, the creation of the clone of `a` in advance by `c` is inadequate. Therefore, in $\textsc{Pan}_{\mathrm{mov}}$, the clone is created by the boundary forwarder when it receives the completion message from `root`. In the proposed execution model, the boundary forwarder belongs to *both* network domains; hence, it can create the clone within the domain below itself. As an alternative, it would be possible to create the clone when the $\{\overline{\mathtt{in}}\}$ message arrives at the boundary forwarder. However, if no ambient executes the corresponding `in` action, the created clone will become unnecessary; this is undesirable.

After the movement, the state finally becomes:

$$\mathtt{root}[\,\rhd^{\bullet}\{\,\mathtt{c}[\,R \mid \mathtt{a}'[\,P \mid\ \rhd^{\bullet}\{\,\rhd^{\circ}\{\,\mathtt{b}[\,Q\,]\,\}\,\}\,]\,]\,\}\,]$$

The newly created boundary forwarder on the right represents the same network boundary as that on the left; however, it forwards messages in the opposite direction.

3.2 Formal Definition

In this section, we formalize the proposed abstract machine $\textsc{Pan}_{\mathrm{mov}}$ by the set of reduction rules for *network configurations*. The definition method basically follows that of Pan [12].

First, a network configuration of $\textsc{Pan}_{\mathrm{mov}}$ is represented using the following syntax:

Nets

$$\begin{array}{rcl} A & ::= & \mathbf{0} \mid \mathit{Agent} \mid \mathit{Msg} \mid A_1 \parallel A_2 \mid (\boldsymbol{\nu} p)A \\ \mathit{Agent} & ::= & h : n[P]_k \mid h \rhd^{B} k \mid h \bullet k, \qquad B ::= \circ \mid \bullet \end{array}$$

where $n \in \mathit{Names}$, $h, k \in \mathit{Locations}$, and $p \in \mathit{Names} \cup \mathit{Locations}$. The overall network A consists of parallel compositions ($\parallel$) of *Agent*s and *Msg*s. There are three kinds of agents: $h : n[P]_k$ is a located ambient mentioned earlier, $h \rhd^{B} k$ is a forwarder at h, which forwards messages to k, and $h \bullet k$ is a seed of a boundary forwarder at h (the meaning of k is explained later). B in a forwarder indicates whether it is a normal forwarder ($\circ$) or boundary forwarder ($\bullet$).

Messages

$$\begin{aligned} \mathit{Msg} &::= \uparrow_h^k\{\mathit{Req}\} \mid \uparrow^h\{\mathit{Compl}\}, \qquad \mathit{Req} ::= R \mid {}_\bullet R \\ R &::= \texttt{in}\ n,m \mid \overline{\texttt{in}}\ n,h \mid \texttt{out}\ n,m \mid \overline{\texttt{open}}\ n \\ \mathit{Compl} &::= \texttt{go}\ h \mid \texttt{OK}\overline{\texttt{in}} \mid \texttt{migrate}\ h \mid \texttt{register}\ P \mid \texttt{new}\ n,k \\ &\quad \mid {}_\bullet\texttt{Cin}\ n,m,h,k \mid {}_\bullet\texttt{Cout}\ n,h \end{aligned}$$

$\uparrow_h^k\{\mathit{Req}\}$ represents a request message sent from h to k and $\uparrow^h\{\mathit{Compl}\}$ represents a completion message sent to h. When a request message crosses at least one boundary, it is marked with the tag $\bullet$, e.g., $\uparrow_h^k\{{}_\bullet R\}$. The meaning of each kind of R and Compl is described in detail below.

Processes

$$\begin{aligned} P ::= {} & \mathbf{0} \mid P_1 \mid P_2 \mid (\boldsymbol{\nu} n)P \mid M.P \mid M[P] \mid \texttt{split}\ M[P] \mid \langle M \rangle \\ & \mid (x)P \mid X \mid \texttt{rec}\ X.P \mid \texttt{wait}.P \mid \uparrow_h\{\mathit{Req}\} \\ M ::= {} & x \mid n \mid \texttt{in}\ M \mid \overline{\texttt{in}}\ M \mid \texttt{out}\ M \mid \overline{\texttt{out}}\ M \mid \texttt{open}\ M \mid \overline{\texttt{open}}\ M \end{aligned}$$

The syntax of the processes is nearly similar to that of SA; the three additional constructs are a cross-boundary ambient creation, $\texttt{split}\ M[P]$, a process waiting for the arrival of any message, $\texttt{wait}.P$, and a request message arriving at its destination ambient, $\uparrow_h\{\mathit{Req}\}$.

The operational semantics of PAN_{mov} is defined as the reduction relation $\longmapsto$ between network configurations. In addition, to express process-level reductions, we use another form of reduction relation, $P \xmapsto[h:n]{k} Q \gg \mathit{Msg}$, to indicate that a process P, local to an ambient n that is located at h, and whose parent is located at k, becomes Q, and the message Msg is emitted as a side effect.

First, the inference rules for $\longmapsto$ are defined as follows:

Inference rules

$$[\text{Proc-Agent}]\ \frac{P \xmapsto[h:n]{k} P' \gg M \qquad Q \text{ has no unguarded ambient}}{h : n[P \mid Q]_k \longmapsto h : n[P' \mid Q]_k \parallel M}$$

$$[\text{Par-Agent}]\ \frac{A_1 \longmapsto {A_1}'}{A_1 \parallel A_2 \longmapsto {A_1}' \parallel A_2} \qquad [\text{Res-Agent}]\ \frac{A \longmapsto A'}{(\boldsymbol{\nu} p)A \longmapsto (\boldsymbol{\nu} p)A'}$$

$$[\text{Struct-Cong}]\ \frac{A \equiv A' \quad A' \longmapsto A'' \quad A'' \equiv A'''}{A \longmapsto A'''}$$

The rule [Proc-Agent] embeds a process-level reduction step into $\longmapsto$. The side condition about Q ensures that all child ambients of n are activated before any local process-level reduction occurs. The remaining rules are straightforward inference rules about contexts and structural congruence. The definition of structural congruence $\equiv$ is mostly standard; hence, it is omitted.

The other axiomatic rules are classified into six categories according to the stages of ambient interactions. In these rules, when some fields or variables are unimportant, we replace them with $-$.

Creation

$$\text{[New-Locamb]} \quad h : m[n[P] \mid Q]_k \longmapsto h : m[Q]_k \parallel (\boldsymbol{\nu} l)(l : n[P]_h), l \notin FL(P)$$

$$\text{[New-Locamb}'\text{]} \quad h : m[\texttt{split}\ n[P] \mid Q]_k \longmapsto h : m[Q]_k \parallel (\boldsymbol{\nu} l)(l \rhd^{\bullet} h \parallel (\boldsymbol{\nu} l')(l' : n[P]_l)),\ l, l' \notin FL(P)$$

$$\text{[New-Res]} \quad h : m[(\boldsymbol{\nu} n)P]_k \longmapsto (\boldsymbol{\nu} n)(h : m[P]_k),\ m \neq n$$

In [New-Locamb], an ambient n is created at the fresh location l, whose parent is located at h. In [New-Locamb′], a new boundary forwarder is also created and inserted between m and n. [New-Res] creates a globally unique name for each name restriction.

Emission of request messages

$$\text{[Req-In]} \quad \texttt{in}\ m.P \xmapsto[h:n]{k} \texttt{wait}.P \gg \uparrow_h^k \{\texttt{in}\ m, n\}$$

$$\text{[Req-Coin]} \quad \overline{\texttt{in}}\ n.P \xmapsto[h:n]{k} \texttt{wait}.P \gg \uparrow_h^k \{\overline{\texttt{in}}\ n, h\}$$

$$\text{[Req-Out]} \quad \texttt{out}\ m.P \xmapsto[h:n]{k} \texttt{wait}.P \gg \uparrow_h^k \{\texttt{out}\ m, n\}$$

$$\text{[Req-Coopen]} \quad \overline{\texttt{open}}\ n.P \xmapsto[h:n]{k} \texttt{wait}.P \gg \uparrow_h^k \{\overline{\texttt{open}}\ n\}$$

These rules are straightforward. For each action or coaction listed above, an ambient sends the corresponding request message to its parent at k. The name n of the requesting ambient is included in the $\{\texttt{in}\}$ and $\{\texttt{out}\}$ message so that it can be used for creating a clone of the ambient if the request crosses a boundary. Note that every single-threaded (ST) ambient that sends a request message to its parent simply blocks waiting for any completion message to be sent back from the parent. This fairly simplifies the execution of processes within each ambient.

Transmission of request messages

$$\text{[Fw-Req]} \quad h \rhd^{\circ} k \parallel \uparrow_l^h \{Req\} \longmapsto h \rhd^{\circ} k \parallel \uparrow_l^k \{Req\}$$

$$\text{[BFw-Req]} \quad h \rhd^{\bullet} k \parallel \uparrow_l^h \{\_R\} \longmapsto h \rhd^{\bullet} k \parallel (\boldsymbol{\nu} h')(h' \bullet l \parallel \uparrow_{h'}^k \{{}_{\bullet}R\})$$

$$\text{[Loc-Rcv]} \quad h : n[P]_k \parallel \uparrow_l^h \{Req\} \longmapsto h : n[P \mid \uparrow_l \{Req\}]_k$$

In [Fw-Req], if a request message reaches a normal forwarder, it is forwarded to k by this normal forwarder. Note that the source location of the message remains l so that the corresponding completion message can be directly sent back to l at the next stage. In [BFw-Req], if a request message reaches a boundary forwarder, it is forwarded to k after being marked with special tag $\bullet$. Furthermore, the source location of the messages is replaced by the fresh location h', where a new seed attached with l is created so that the corresponding completion message can be sent back via this seed. In [Loc-Rcv], when a request message reaches its destination, it is brought into the destination.

Local reductions

$$[\textsc{Local-Com}] \quad \langle M\rangle \mid (x)P \xrightarrow[-:-]{-} P\{M/x\} \gg \mathbf{0}$$

$$[\textsc{Local-In}] \quad \uparrow_l\{\mathtt{in}\ n,-\} \mid \uparrow_{l'}\{\overline{\mathtt{in}}\ n,l'\} \xrightarrow[-:-]{-} \mathbf{0} \gg \uparrow^l\{\mathtt{go}\ l'\} \parallel \uparrow^{l'}\{\mathtt{OK}\overline{\mathtt{in}}\}$$

$$[\textsc{Local-In}'] \quad \uparrow_l\{\_\mathtt{in}\ n,m\} \mid \uparrow_{l'}\{\bullet\overline{\mathtt{in}}\ n,k\} \xrightarrow[-:-]{-} \mathbf{0} \gg \uparrow^l\{\mathtt{migrate}\ l'\} \parallel \uparrow^{l'}\{\mathtt{new}\ m,k\}$$

$$[\textsc{Local-In}''] \quad \uparrow_l\{\bullet\mathtt{in}\ n,m\} \mid \uparrow_{l'}\{\overline{\mathtt{in}}\ n,l'\} \xrightarrow[h:-]{-} \mathtt{wait}.\mathbf{0} \gg \uparrow^h\{\bullet\mathtt{Cin}\ n,m,l,l'\}$$

$$[\textsc{Local-Out}] \quad \uparrow_l\{\mathtt{out}\ n,-\} \mid \overline{\mathtt{out}}\ n.P \xrightarrow[-:n]{k} P \gg \uparrow^l\{\mathtt{go}\ k\}$$

$$[\textsc{Local-Out}'] \quad \uparrow_l\{\bullet\mathtt{out}\ n,m\} \mid \overline{\mathtt{out}}\ n.P \xrightarrow[h:n]{-} \mathtt{wait}.P \gg \uparrow^h\{\bullet\mathtt{Cout}\ m,l\}$$

$$[\textsc{Local-Open}] \quad \mathtt{open}\ n.P \mid \uparrow_l\{\_\overline{\mathtt{open}}\ n\} \xrightarrow[h:-]{-} \mathtt{wait}.P \gg \uparrow^l\{\mathtt{migrate}\ h\}$$

[Local-Com] is the same as [R-Msg] of SA in Section 2.1. The other rules express match operations of an ambient for three kinds of movements.

In [Local-In] and [Local-Out], if no request message is marked with the $\bullet$ tag, the appropriate completion messages are sent back, as shown in Figure 1 (a), (b). In [Local-Open], irrespective of the $\{\overline{\mathtt{open}}\}$ message being marked with $\bullet$, all opens can be handled as shown in Figure 1 (c).

If an $\{\overline{\mathtt{in}}\}$ message sent by an ambient at k is marked with $\bullet$, a clone must be created inside the same network domain as the ambient, as explained in Section 3.1. The message $\{\mathtt{new}\ m,k\}$ is used for this purpose in [Local-In$'$], where m is the name of the clone to be created.

On the other hand, if an $\{\overline{\mathtt{in}}\}$ message does not cross any boundaries, a clone may be created immediately by the parent. However, in our formalization, we cannot express the creation of a located agent at the process level. Instead, we formalized this case as in [Local-In$''$], where the parent sends the $\{\bullet\mathtt{Cin}\}$ message to *itself*. The consumption of $\{\bullet\mathtt{Cin}\}$ is a network-level reduction (see [Compl-Cin] below); hence, it can express the creation of a clone. We do the same for `out` in [Local-Out$'$]. [1]

Transmission of completion messages

$$[\textsc{Back-Migr}] \quad \uparrow^h\{\mathtt{migrate}\ k\} \parallel h \bullet l \longmapsto h \rhd^\bullet k \parallel \uparrow^l\{\mathtt{migrate}\ h\}$$

$$[\textsc{Back-New}] \quad \uparrow^h\{\mathtt{new}\ m,k\} \parallel h \bullet l \longmapsto h \rhd^\bullet l \parallel \uparrow^l\{\mathtt{new}\ m,k\},\ k \neq l$$

$$[\textsc{Back-New}'] \quad \uparrow^h\{\mathtt{new}\ m,l\} \parallel h \bullet l \longmapsto (\boldsymbol{\nu} k)(h \rhd^\bullet k \parallel k : m[\mathtt{wait}.\mathbf{0}]_l \parallel \uparrow^l\{\mathtt{OK}\overline{\mathtt{in}}\})$$

The kinds of completion messages that might cross some network boundary are `new` and `migrate`; both are forwarded by seeds.

In [Back-Migr], the seed at h changes the argument of $\{\mathtt{migrate}\ k\}$ to h and forwards it to l, where a requesting ambient or another boundary forwarder is located. At the same time, the seed becomes the boundary forwarder in preparation for the $\{\mathtt{register}\}$ message sent back from l.

[1] These extra steps (local communication) can be omitted in real implementations. In addition, the parameter n of $\{\bullet\mathtt{Cin}\ n,m,l,l'\}$ is used only for the correctness proof (see Section 6 and [6]).

In [Back-New], the condition $k \neq l$ implies that the agent at l is not the ambient that requested $\overline{\texttt{in}}$; it is another boundary forwarder. Thus, the seed simply forwards the $\{\texttt{new}\}$ message to l, and then, it becomes the boundary forwarder in preparation for the $\{\texttt{register}\}$ message sent from *elsewhere*. This boundary forwarder must forward it to l because the new clone will be created beyond l.

In [Back-New′], when a $\{\texttt{new}\}$ message eventually reaches the same network domain as the ambient that requested $\overline{\texttt{in}}$ at l, a clone is created at the fresh location k and the seed at h becomes the boundary forwarder targeting k. Furthermore, an $\texttt{OK}\overline{\texttt{in}}$ message, which notifies the match of $\overline{\texttt{in}}$, is sent to l.

Consumption of completion messages

[Compl-Parent] $\uparrow^h \{\texttt{go}\ k\} \parallel\ h : n[P \mid \texttt{wait}.Q]_- \longmapsto h : n[P \mid Q]_k$

[Compl-Coin] $\uparrow^h \{\texttt{OK}\overline{\texttt{in}}\} \parallel\ h : n[P \mid \texttt{wait}.Q]_k \longmapsto h : n[P \mid Q]_k$

[Compl-Cin] $\uparrow^h \{\bullet\texttt{Cin}\ -, m, l, l'\} \parallel\ h : n[P \mid \texttt{wait}.Q]_k \longmapsto$
$h : n[P \mid Q]_k \parallel\ (\boldsymbol{\nu} h')(h' : m[\texttt{wait}.\mathbf{0}]_{l'} \parallel\ \uparrow^l \{\texttt{migrate}\ h'\}) \parallel\ \uparrow^{l'} \{\texttt{OK}\overline{\texttt{in}}\}$

[Compl-Cout] $\uparrow^h \{\bullet\texttt{Cout}\ m, l\} \parallel\ h : n[P \mid \texttt{wait}.Q]_k \longmapsto$
$h : n[P \mid Q]_k \parallel\ (\boldsymbol{\nu} h')(h' : m[\texttt{wait}.\mathbf{0}]_k \parallel\ \uparrow^l \{\texttt{migrate}\ h'\})$

[Compl-Migr] $\uparrow^h \{\texttt{migrate}\ k\} \parallel\ h : n[P \mid \texttt{wait}.Q]_- \longmapsto$
$h \rhd^\circ k \parallel\ \uparrow^k \{\texttt{register}\ P \mid Q\}$

[BFw-Reg] $\uparrow^h \{\texttt{register}\ R\} \parallel\ h \rhd^\bullet k \longmapsto$
$h \rhd^\bullet k \parallel\ \uparrow^k \{\texttt{register}\ R\}$

[Compl-Reg] $\uparrow^h \{\texttt{register}\ R\} \parallel h : n[P \mid \texttt{wait}.Q]_k \longmapsto h : n[P \mid Q \mid R]_k$

In [Compl-Parent] and [Compl-Coin], $\{\texttt{go}\}$ and $\{\texttt{OK}\overline{\texttt{in}}\}$ messages are handled appropriately by the destination ambient. In [Compl-Cin] and [Compl-Cout], an ambient that receives a $\{\bullet\texttt{Cin}\}$ or $\{\bullet\texttt{Cout}\}$ message creates a clone and sends the appropriate completion messages. In [Compl-Migr], an ambient that receives a $\{\texttt{migrate}\ k\}$ message sends the $\{\texttt{register}\}$ message, which contains the contents of the ambient, to k. Each $\{\texttt{register}\}$ message reaches the destination ambient through zero or more transmissions of [BFw-Reg]; then, R is merged into the destination in [Compl-Reg].

4 GcPan$_{\text{mov}}$: Garbage Collecting Forwarders

As described in the previous section, chaining forwarders upon each cross-boundary movement in Pan$_{\text{mov}}$ removes the need for all-to-all connectivity in the underlying network. However, along with this adaptability to wide-area networks, at least one boundary forwarder and one normal forwarder are created upon each cross-boundary movement. Therefore, continuing the execution of an SA program in Pan$_{\text{mov}}$ is likely to result in the accumulation of more unused forwarders and longer chains of forwarders than those in Pan. Clearly, unused forwarders keep occupying resources needlessly, and forwarder chains induce a loss of performance by increasing the number of network messages.

To handle such situations, we enriched PAN$_{\mathrm{mov}}$ with the mechanism used in GCPAN [8]; it reclaims unused forwarders and contracts forwarder chains. The following are the basic ideas of GCPAN: (1) to detect unused forwarders, every agent is equipped with a reference count. Every time an agent receives a request message, its reference count is decremented. If an agent is a forwarder whose count is zero, it is reclaimed, and (2) to contract forwarder chains, using Tarjan's union-find algorithm [14], every agent is relocated immediately below its parent ambient. For a detailed understanding of GCPAN, see [8].

Enriching PAN$_{\mathrm{mov}}$ with GCPAN's mechanism is a straightforward process. However, the formal definition of the resulting abstract machine GCPAN$_{\mathrm{mov}}$ is rather complex. For lack of space, it is provided in [6].

5 GCPAN$_{\mathbf{shift}}$: Proactive Movement

In PAN$_{\mathrm{mov}}$ and GCPAN$_{\mathrm{mov}}$, an ambient physically moves when it performs a cross-boundary `in` or `out` action. However, this implies that its physical movement is still deferred until it sends some request message to its parent. Then, for instance, if an ambient blocks for some reason, e.g., waiting for I/O, forwarders used by the ambient will not be reclaimed or contracted until the ambient resumes and sends some request. Moreover, if a network connection represented by a certain network boundary is broken for some reason (including permanent disconnection of mobile devices), a child ambient whose parent belongs to the other side of the boundary cannot send requests to its parent, and it cannot perform movements anymore.

In order to address the problem described above, there should be a mechanism for enforcing physical movements of ambients in the abstract machines, which PAN and GCPAN do not have. Therefore, we added a `shift` action, which instructs an ambient to move physically below its parent, in PAN$_{\mathrm{mov}}$ and GCPAN$_{\mathrm{mov}}$. Note that `shift` performs no logical action at the calculus level; hence, each ambient may perform `shift` actions periodically. For example, in the state:

$$h_1 : \mathtt{a}[\,P\,]_{h_2} \parallel h_2 \rhd^{\circ} h_3 \parallel h_3 \rhd^{\bullet} h_4 \parallel h_4 \rhd^{\circ} h_5 \parallel h_5 : \mathtt{b}[\,Q\,]_{h_6}$$

if `a` performs a `shift` action, the state changes to:

$$h_2 \rhd^{\circ} h_3 \parallel h_3 \rhd^{\bullet} h_4 \parallel h_4 \rhd^{\circ} h_5 \parallel h_7 : \mathtt{a}[\,P\,]_{h_5} \parallel h_5 : \mathtt{b}[\,Q\,]_{h_6}$$

Moreover, if `a` has no child or if all of `a`'s children also perform a `shift` action, the forwarders are reclaimed as in: $h_7 : \mathtt{a}[\,P\,]_{h_5} \parallel h_5 : \mathtt{b}[\,Q\,]_{h_6}$.

We enrich PAN$_{\mathrm{mov}}$ with the `shift` action described above by adding the following reduction rules:

$$\begin{array}{lll}
\text{[REQ-SHIFT]} & P \xmapsto[h:n]{k} \mathtt{wait}.P & \gg \uparrow_h^k \{\mathtt{shift}\ n\} \\
\text{[LOCAL-SHIFT]} & \uparrow_l \{\mathtt{shift}\ -\} \xmapsto[h:-]{-} \mathbf{0} & \gg \uparrow^l \{\mathtt{go}\ h\} \\
\text{[LOCAL-SHIFT}'\text{]} & \uparrow_l \{\mathtt{shift}_{\bullet}\ m\} \xmapsto[h:-]{-} \mathtt{wait}.\mathbf{0} & \gg \uparrow^h \{\mathtt{Cshift}_{\bullet}\ m, l\}
\end{array}$$

[Compl-CShift] $\uparrow^h \{\texttt{Cshift}_\bullet\ m, l\} \parallel\ h : n[P \mid \texttt{wait}.Q]_k \longmapsto$
$h : n[P \mid Q]_k \parallel\ (\boldsymbol{\nu} h')(h' : m[\texttt{wait}.\mathbf{0}]_h \parallel\ \uparrow^l \{\texttt{migrate}\ h'\})$

The formal definition of the abstract machine GcPAN$_{\text{shift}}$, which is an extension of GcPAN$_{\text{mov}}$ with the `shift` action, is provided in [6].

6 Correctness

We prove that PAN$_{\text{mov}}$ is a correct implementation of SA. The fact that PAN is a correct implementation of SA, i.e., SA and PAN are weak barbed bisimilar, is proved in [12, 7]; hence, it suffices to prove that PAN and PAN$_{\text{mov}}$ are weak barbed bisimilar. We follow the proof method between PAN and GcPAN [8, 9].

Theorem 1. *There is a weak barbed bisimulation $\mathcal{R}$ between* PAN *and* PAN$_{\text{mov}}$.

Proof. We constructed such a bisimulation relation $\mathcal{R}$. For lack of space, the precise definition of $\mathcal{R}$ and the details of the proof is provided in [6].

Due to physical movement of PAN$_{\text{mov}}$, $\mathcal{R}$ has several significant differences from the bisimulation relation $\mathcal{R}'$ between PAN and GcPAN, which is described in [9]. The following are the main differences: (1) each ambient in PAN$_{\text{mov}}$ may be at several locations during its lifetime; therefore, the simple correspondence between ambients that are at the same location, which is used in $\mathcal{R}'$, does not work. Instead, we established a mapping from the set of ambient locations in a PAN net to the set of ambient locations in the corresponding PAN$_{\text{mov}}$ net, and (2) the mapping is updated along with PAN$_{\text{mov}}$'s reductions related to physical movement of ambients so that it can properly map unmoving ambients of PAN to moving ambients of PAN$_{\text{mov}}$. □

Corollary 2 (Adequacy). *Let P be an SA process, then $[\![P]\!]_{mov} \approx P$.*

Proof. $[\![P]\!]\ \mathcal{R}\ [\![P]\!]_{mov}$ can be easily checked; hence, from Theorem 1, $[\![P]\!] \approx [\![P]\!]_{mov}$. Then, the result follows from the adequacy of PAN [7]: $[\![P]\!] \approx P$. □

For the correctness of GcPAN$_{\text{shift}}$, the proof stated above can be adapted to establish a weak barbed bisimilarity between GcPAN$_{\text{shift}}$ and GcPAN. The correctness of GcPAN$_{\text{mov}}$ is immediately derived from that of GcPAN$_{\text{shift}}$ because GcPAN$_{\text{mov}}$ is a subset of GcPAN$_{\text{shift}}$.

7 Conclusion

In this paper, we proposed novel abstract machines that can handle network boundaries in wide-area and mobile networks. They have the following desirable properties: (1) No ambients communicate directly with other ambients in different network domains; instead, all inter-domain messages are sent via boundary forwarders, and (2) In GcPAN$_{\text{shift}}$, any ambient will go into a stable state in which any ambient inside it can perform SA actions without boundary forwarders. Therefore, we can construct more reliable implementations of SA by

using these machines as base implementation models. Formal proofs of these properties are left for future work.

AtJ [4] is a distributed implementation of AC, a translator from AC to JoCaml [3]. Although physical movement is triggered by each execution of `in` or `out`, no forwarders are created for this movement in AtJ. This is because any child can send messages directly to its parent at any moment using JoCaml's distributed message transfer mechanism; that is, AtJ relies on JoCaml's all-to-all connectivity. Nonetheless, adapting our technique to AtJ seems relatively easy because a forwarder is created at each `open` action, as in the case of PAN.

In a distributed abstract machine for the Kell Calculus [13], the passivation of a kell is represented as the physical migration of the *whole* hierarchy (i.e., the kell, its sub-kells, sub-kells of its sub-kells, and so on). Thus, the underlying network need not support all-to-all connectivity. However, such a passivation mechanism seems rather inefficient. Moreover, each kell must keep track of its sub-kells; therefore, the abstract machine is more complex, as compared to the PAN family.

Acknowledgments. This work was partly supported by MEXT Grant-in-Aid for Young Scientists (B) (21700029).

References

1. Cardelli, L.: Mobile Ambient Synchronization. Technical Report 1997-013, Digital Systems Research (1997)
2. Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Proc. of FoSSaCS'98, LNCS 1378, 140–155 (1998)
3. Fournet, C.: The Join-Calculus: a Calculus for Distributed Mobile Programming. PhD thesis, Ecole Polytechnique (1998)
4. Fournet, C., Lévy, J.J., Schmitt, A.: An Asynchronous, Distributed Implementation of Mobile Ambients. In: Proc. of IFIP TCS'00, LNCS 1872, 348–364 (2000)
5. GCPAN webpage: `http://perso.ens-lyon.fr/damien.pous/gcpan`.
6. GCPAN$_{\text{shift}}$ webpage: `http://ryujin.kuis.kyoto-u.ac.jp/~umatani/pan/`.
7. Giannini, P., Sangiorgi, D., Valente, A.: Safe Ambients: Abstract Machine and Distributed Implementation. Science of Computer Programming **59**, 209–249 (2006)
8. Hirschkoff, D., Pous, D., Sangiorgi, D.: A Correct Abstract Machine for Safe Ambients. In: Proc. of COORDINATION 2005, 17–32 (2005)
9. Hirschkoff, D., Pous, D., Sangiorgi, D.: An efficient abstract machine for Safe Ambients. Journal of Logic and Algebraic Programming **71**(2), 114–149 (2007)
10. Levi, F., Sangiorgi, D.: Mobile Safe Ambients. ACM Transactions on Programming Languages and Systems **25**, 1–69 (2003)
11. Phillips, A., Yoshida, N., Eisenbach, S.: A Distributed Abstract Machine for Boxed Ambient Calculi. In: Proc. of ESOP'04, LNCS, Springer, 155–170 (2004)
12. Sangiorgi, D., Valente, A.: A Distributed Abstract Machine for Safe Ambients. Automata, Languages and Programming, LNCS 2076, 408–420 (2001)
13. Schmitt, A., Stefani, J.: An Abstract Machine for the Kell Calculus. In: Proc. of IFIP FMOODS 2005, 43–58 (2005)
14. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the ACM **22**, 215–225 (1975)