An Algorithm for Distributed Location Management in Networks of Mobile Computers*

Svetlana Kryukova†, Berna Massingill‡, and Beverly Sanders§

Abstract

In a network supporting mobile communication devices, a mechanism to find the location of a device, wherever it may be, is needed. In this paper, we present a distributed algorithm for this purpose along with its formal specification and proof sketch. Inspired by our experiences with Wang's algorithm [9], one goal of this paper is to demonstrate that the process of formalization together with careful attention to abstraction and presentation can yield significant benefits in algorithm design. In this case, we obtained a more regular, general, and robust algorithm with a clearer description. An incidental contribution is a useful theorem for proving progress properties in distributed algorithms that use tokens.

1 System Description

We consider a system with a fixed network consisting of a set of *nodes* with unique IDs and communication links between them, plus a large number of mobile devices we will call *portables*. Routing[1] between each pair of nodes in the fixed network is provided; knowing the identity of a node is sufficient to be able to communicate with it.

Each portable has a unique ID and is associated with a node called its *home address*. Given the portable's ID, it is possible to determine the home address. We will denote the home address for portable P as $HA.P$. At any given time, a portable may be associated with a unique node of the fixed network with which it communicates directly, typically over a wireless link. The ID of this node is

*This work was supported in part by the AFOSR under grant number AFOSR-91-0070 and in part by the University of Florida.

†Tanner Research, Inc., 180 North Vinedo Ave., Pasadena, CA 91107, kryukova@tanner.com.

‡Department of Computer Science, California Institute of Technology, m/c 256-80, Pasadena, CA 91125, berna@cs.caltech.edu.

§Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120, sanders@cis.ufl.edu.

[1] Routing in fixed networks has several well known solutions. See, for example, [8].

called the portable's location or *actual address*. The actual address of portable P is denoted $AA.P$. A message can be sent to the portable via its actual address node, which may change over time as the portable moves. Each portable is also associated with a subset of nodes called its *service area*. The service area models the region in which the system should be able to locate the portable.

In this paper we give algorithms for maintaining and querying a database containing location information for portables. A query to the database returns the actual address of the portable or indicates that the portable is out of its service area or is switched off.

Although the actual address of a portable while it is communicating will correspond to a base station[2], in modeling the system we have abstracted away from the specific devices that make up the fixed network. Indeed, we don't even assume a one-to-one correspondence between devices and nodes. For example, in order to provide fault tolerance, a node may consist of several different physical devices that, from the point of view of our algorithms, form a single logical entity[3]. On the other hand, since the system tracks when the portable is unreachable due to being out of its service area or switched off, we introduce virtual nodes representing the (fictitious) "location" of portables that are switched off or out of their service areas, and then require that each portable be associated with exactly one node at all times. This allows being switched on or off and moving in and out of the service area to be handled using the same mechanism as other movement of portables. The special nodes representing the location of an inactive portable with home address S would be implemented by the same physical device(s) as node S.

We have also abstracted away from the particular way that nodes and portables are named and routing among the fixed network is carried out. Typically, the IDs of nodes and of portables will contain geographic information similar to telephone numbers that can be exploited for routing and determining a node's home address. See [7, 9] for a discussion of numbering and addressing in mobile networks. We assume that routing in the fixed network is available, and that given a portable's ID there is some way to determine its home address. We do not specify exactly how this is done, since it is not relevant to our algorithms.

The algorithms are developed and verified using a variation of the UNITY [1, 4, 5] programming method. We have attempted to give sufficient intuitive justification and informal explanations to allow readers unfamiliar with this formalism to understand the algorithms and follow the overall derivation. Appendix A supplies necessary definitions, axioms, and theorems from UNITY.

[2] A base station is a node of the physical network that also engages in wireless communication with portables, acting as the interface between portables and the physical network.

[3] Several known techniques exist for providing fault tolerance using replication [2].

2 Problem Description

In terms of our model, the problem is to give algorithms that allow the current actual address of each portable in the system, or the fact that it is unreachable (due to being out of its service area or switched off), to be determined. We will give one algorithm for database maintenance and one for database queries.

The GSM and EIA/TIA standards for mobile communications (see [6] for an overview of location strategies in both) implement a two-level database of location information. For portables away from their home addresses, the home address node maintains the location of a "visitor location register" that stores the actual address of the portable. Any location change that requires a change in the visitor location register requires that an update message be sent to the home address node. The actual address of a portable is obtained by sending a message to its home address node, which forwards it to the visitor location register. This approach has the disadvantage that it doesn't scale well, and in cases where the portable is not close (in the network) to the home address node it can result in a significant amount of communication.

A more distributed approach, described in [9], imposes a hierarchical tree structure on the nodes in the fixed network. Wang calls this an "intelligent network". In Wang's algorithm, the length of the path from the root of the tree to each leaf node is constant, and each level in the tree represents a geographic area. For example, the top level is the world, the nodes at the next level represent countries, the next level states, and so on. Leaf nodes correspond to (clusters of) base stations that can communicate with portables. The node addressing scheme is based on this structure. For each portable away from its home address, the algorithm determines a path from the portable's home address to its current location and stores information about this path in the nodes that comprise it. Together, these paths make up a distributed location-registration database.

He gives simulation results showing that better performance can be obtained with this scheme than with the proposed EIA/TIA standard, since the hierarchical scheme results in less message traffic and lower latency.

Our algorithm is more abstract and general than Wang's in that we logically structure the network as an arbitrary undirected connected acyclic graph (rather than as a tree of fixed depth), and we allow an actual address to be any node in this graph. This abstraction is largely independent of the physical topology of the network. This generalization allows a more regular algorithm since it enables us to handle portables that are switched off or out of their service areas with the same mechanism used for other location changes.

The relevant property of a connected acyclic graph is that there is a unique shortest path along the edges of the graph between any two nodes. Most of the messages used by the algorithm are sent along edges of the acyclic graph.

The actual address of a portable is represented by its *location path*: the path in the acyclic graph from the home address node to the actual address of the portable. The location of a portable can be determined by traversing the acyclic graph toward the portable's home address node until the portable's location path is intersected and then following this path to the actual address node of the portable. The path followed by the query will intersect the location path at the home address, if not sooner. The path from the home address to the portable must be updated as the portable moves in such a way that the portable can eventually be reached, even if its address is sought while it is moving (provided the portable doesn't move too fast).

3 System and Component Models

In this section, we give a more formal description of the system and component models and remark on notation.

3.1 Portables

For a portable P, in addition to the home address $HA.P$, the actual address $AA.P$, and the service area, all of which were discussed in section 1, we associate with portable P a logical clock, denoted $ts.P$. This clock could be implemented with a physical clock or counter on the portable itself, or as a sufficiently synchronized global time implemented by the nodes of the fixed network. The necessary property of the clock is that its value increases (at least) on movement of the portable to a new location. This is formally specified with the following property:

$$ts.P = k \land AA.P = S \quad next \quad (ts.P \geq k \land AA.P = S) \lor ts.P > k \tag{1}$$

3.2 The fixed network

The fixed network is modeled as a set of nodes organized in an undirected connected acyclic graph whose important characteristic is that there is a unique shortest path between every pair of nodes. The function $minPath : (node \times node) \rightarrow list\ of\ node$ is given and fixed for any particular network. In particular, $minPath(N, M)$ is the list of nodes corresponding to the shortest path between nodes N and M, not including N, and $path(N, N) = \langle\rangle$, the empty list. The reason for not including N in the path is so that the concatenation $minPath(N, M) +\!+\, minPath(M, R)$ makes sense. The list so obtained is a path over the acyclic graph from N to R, although not necessarily the shortest one (since it could contain a cycle). If it is desired to have a path representation that includes N, this can be indicated using the cons operator "::" (e.g., $N :: minPath(N, M)$). Although we will often use the $minPath$ function in specifications, in the algorithms themselves a node will only need to determine the next node on a path from itself to some other node. For convenience, we define the function $nextNode$ to return the next node.

$$nextNode(N, M) \stackrel{\text{def}}{=} head(minPath(N, M)) \tag{2}$$

We observe that since $minPath(N, N) = \langle\,\rangle$, $nextNode(N, N)$ is undefined.

3.3 Communication and computation

Nodes communicate with each other via asynchronous message-passing. After a message is sent, it will eventually be available to be received by its destination. At each step, a node may read a message, compute and modify local variables, and send one or more messages. Read, compute, and send are performed in a single atomic step, and we require that all messages that are sent are eventually received. We assume that routing is provided for communication between any two points in the fixed network. In our algorithm, however, most of the communication will be between nodes that are neighbors in the acyclic graph.

The algorithms for maintaining and querying the database are based on tokens. Tokens are simply message records in which the first three fields are message ID, sender, and destination. Additional fields indicating the token type and containing additional data are also used. The predicate $token$ holds when a particular token is in the system, i.e.,

$$token(P, R, S, type, data) \stackrel{\text{def}}{\equiv} \text{a message } (P, R, S, type, data) \text{ is in transit between nodes } R \text{ and } S \tag{3}$$

We use underscores as an abbreviation for existential quantification:

$$token(P, \_, \_, type, data) \stackrel{\text{def}}{=} (\exists r, s :: token(P, r, s, type, data)) \tag{4}$$

The action of receiving a token $(P, R, S, Type, Data)$ falsifies the predicate $token(P, R, S, Type, Data)$. Sending a token $(P, R, S, Type, Data)$ establishes $token(P, R, S, Type, Data)$. We assume that each message sent will eventually be delivered and processed. With this assumption (discussed in more detail in appendix B), we can write the following leads-to property:

$$token(P, R, S, Type, Data) \leadsto \neg token(P, R, S, Type, Data) \tag{5}$$

Additional details about the communication model are given in appendix B.

3.4 Notation

We sometimes denote application of function f to x with the notation $f.x$. An example of this is the notation used for the home address of portable P: HA is a function mapping portable IDs into node IDs, so $HA.P$ is the node ID that results from applying this function to P's ID. Another example is the use of $AA.P$ to denote the portable's actual address. Here, the function is merely a convenient way to formalize "where the portable is"; moving the portable changes the function.

We use the notation $R.Var$ to denote local variable Var of node R.

4 The database maintenance algorithm

4.1 Overview

As discussed earlier, the idea of the algorithm is to impose on the acyclic graph a path for each portable P from its home address to its actual address. (We call such a path a P-path, and we call its edges P-edges.) Each node R has a variable $R.Out$ that contains a set of (portable ID, node ID) pairs. In other words, $(P, S) \in R.Out$ means that there is an edge labeled P (i.e., a P-edge) from R to S. $R.Out$ can also be thought of as a function on portable IDs, where $R.Out.P$ is the destination of an outgoing P-edge from R, or ϵ if there are none. We will require (in (11)) that there is at most one outgoing P-edge, so $R.Out.P$ is a function, defined thus:

$$R.Out.P \stackrel{\text{def}}{=} \begin{array}{ll} S & \mathit{if}\ (P,S) \in R.Out \\ \epsilon & \mathit{otherwise} \end{array} \tag{6}$$

Also, each node R has a set $R.Registered$ that contains the ID of each portable P for which R is the actual address ($AA.P = R$) and for which the database is currently up to date.

When a portable P with $ts.P = k$ moves from actual address R to actual address T, R learns of the new destination and the value of $ts.P$. P is then removed from $R.Registered$, and a token is created to update the database. The token is sent along the shortest path from R to T, removing or adding edges as necessary to establish a path from $HA.P$ to T. When the token arrives at T, either T is still the actual address of P and P is added to $T.Registered$, or P has moved in the meantime and the update process continues.

In practice, R could be notified of the new destination by receiving a message from the portable, or by discovering that it can no longer communicate with a registered portable and interpreting this as meaning the portable has been turned off. In the latter situation, node R would simply generate an update token and send it to a virtual node representing the "location" of P when it is switched off or out of its service area. Also, note that all actions taken by the database update algorithm are done in response to actions initiated by the old node. This avoids any need to synchronize actions of the old and new nodes.

4.2 Specification

We can specify the database maintenance algorithm as follows. First, if a portable is registered at a node, it should indeed be located there. This is given by an invariant property:

$$P \in T.Registered \;\; \Rightarrow \;\; AA.P = T \tag{7}$$

We also want it to be the case that if a portable's actual address is T, then eventually P will be registered at T, unless P moves to another location before the algorithm has time to finish updating the database. This is given by a leads-to property:

$$AA.P = T \;\; \leadsto \;\; P \in T.Registered \vee AA.P \neq T \tag{8}$$

Our algorithm works by sending a token along a path, so we require that the path be maintained properly. We define $locPath.P$ (P's location path) as the longest sequence of nodes that can be reached by starting at P's home address and following P-edges:

$$locPath.P \;\; \stackrel{\text{def}}{=} \;\; HA.P :: maxPath.P.(HA.P) \tag{9}$$

where

$$maxPath.P.x \;\; \stackrel{\text{def}}{=} \tag{10}$$
$$\begin{array}{cl} \langle\rangle & \textit{if } x.Out.P = \epsilon \\ x.Out.P :: maxPath.P.(x.Out.P) & \textit{otherwise} \end{array}$$

$locPath.P$ is unique and well-defined if each node has at most one outgoing P-edge and no P-edges to itself:

$$(P,S) \in R.Out \;\wedge\; (P,T) \in R.Out \;\; \Rightarrow \;\; S = T \;\wedge\; S \neq R \tag{11}$$

If P is registered at T, we require that there be a P-path from P's home address to T and that this path be the shortest path from $HA.P$ to T. These requirements are formalized with two invariants: The first says that if P is registered at T, then the last node of $locPath.P$ is T. The second says that $locPath.P$ is always the shortest path from $HA.P$ to its last node.

$$P \in T.Registered \;\; \Rightarrow \;\; last(locPath.P) = T \tag{12}$$

$$locPath.P \;\; = \;\; HA.P :: minPath(HA.P, last(locPath.P)) \tag{13}$$

From the properties of acyclic graphs, (13) holds if all nodes in $locPath.P$ are unique.

Also, we don't want "stray" edges, so we require that all P-edges be part of $locPath.P$.

$$R.Out.P = S \;\; \Rightarrow \;\; R \in locPath.P \tag{14}$$

Together, properties (7), (8), (11), (12), (13), and (14) constitute a formal specification of the database maintenance algorithm.

4.3 Algorithm development

Since these properties ((7), (8), (11), (12), (13), and (14)) do not give a strong enough invariant to prove directly, we introduce additional properties to strengthen the specification.

The database update algorithm uses update tokens. An update token has the structure

$$(P, R, S, \mathit{UPDATE}, (t, T)) \tag{15}$$

where P is the portable whose registered address needs to be updated, R is the sender of the token, S is the immediate destination of the token, UPDATE indicates that the token is to update the database, t is a timestamp indicating the time (from $ts.P$) when P moved to T, and T indicates the final destination of the token. At each step, the algorithm receives a token and then forwards it to the next node on the path to T.

We can define several properties about these tokens. First, no update token exists for a portable P exactly when P is registered at its current actual address.

$$\neg token(P, \_, \_, \mathit{UPDATE}, \_) \quad \equiv \quad P \in (AA.P).Registered \tag{16}$$

Also, at any time, there is at most one update token for P.

$$0 \leq \#token(P, \_, \_, \mathit{UPDATE}, \_) \leq 1 \tag{17}$$

Our algorithm guarantees properties (16) and (17) by creating a new update token only when a registered portable becomes unregistered and deleting a token only when registering a portable.

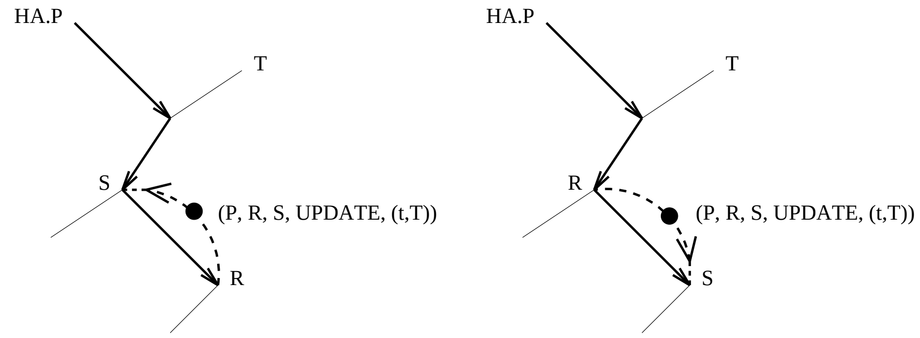
Further, update tokens have the property that a token is always sent along a path to its final argument, which is given by the last field.

$$token(P, R, S, \mathit{UPDATE}, (\_, T)) \quad \Rightarrow \quad S = nextNode(R, T) \tag{18}$$

This property is easily guaranteed.

A new update token is created when a portable moves away from the node where it is registered. From (12), at the moment at which the token, say $(P, R, S, \mathit{UPDATE}, (t, T))$, is created, the last node of P's location path $locPath.P$ is R. The idea of the algorithm is that the update token is at the end of the current $locPath.P$ and moves toward node T, adding or removing edges as it goes. There are two situations: one in which $locPath.P$ needs to be extended to reach T, and one in which edges need to be removed. An update will typically start out removing edges and then later enter a phase in which edges are added. These two situations are shown in figure 1.

As can be seen from the figures, if the token $(P, R, S, \mathit{UPDATE}, (t, T))$ is removing edges, then $last.(locPath.P) = R$; if it is extending $locPath.P$, then $last.(locPath.P) = S$. The two cases can thus be distinguished by comparing $nextNode(R, T)$ (i.e., S) with $nextNode(R, HA.P)$. If they are equal, then



(a) Path contracting. (b) Path extending.

Figure 1: (a) Token $(P, R, S, UPDATE, (t, T))$ removing edges from $locPath.P$. (b) Token $(P, R, S, UPDATE, (t, T))$ extending $locPath.P$.

the token in transit between R and S is moving toward the home address. If $nextNode(R, T) \neq nextNode(R, HA.P)$, then the token is moving away from the home address. Note also that if the token is moving toward the home address, then $S.Out.P = R$, and if it is moving away, then $S.Out.P = \epsilon$. We can state this with the following invariant.

$$
\begin{aligned}
& token(P, R, S, UPDATE, (\_, T)) \;\Rightarrow \\
& \qquad (nextNode(R, T) = nextNode(R, HA.P) \;\Rightarrow \\
& \qquad\qquad (last(locPath.P) = R \;\wedge\; S.Out.P = R)) \\
& \qquad \wedge\; (nextNode(R, T) \neq nextNode(R, HA.P) \;\Rightarrow \\
& \qquad\qquad (last(locPath.P) = S \;\wedge\; S.Out.P = \epsilon))
\end{aligned} \tag{19}
$$

When portable P, registered at R and having $ts.P = t$, moves to T and creates an update token $(P, R, S, UPDATE, (t, T))$, $locPath.P \mathbin{++} minPath(R, T)$ (where $R = last(locPath.P)$) forms a path from P's home address to its actual address T, although not necessarily the shortest one. We define $update.P$ as the shortest path from the last node of $locPath.P$ to the token's final destination:

$$
\begin{aligned}
& update.P \;\stackrel{\text{def}}{=} \\
& \qquad \begin{array}{ll}
minPath(last(locPath.P), T) & \textit{if} \;\; token(P, \_, \_, UPDATE, (\_, T)) \\
\langle\,\rangle & \textit{if} \;\; \neg token(P, \_, \_, UPDATE, \_)
\end{array}
\end{aligned} \tag{20}
$$

As the token moves along, $update.P$ becomes shorter until it is empty while maintaining the invariant that $locPath.P \mathbin{++} update.P$ forms a path, although not necessarily the shortest one, to T. Once $update.P$ becomes empty, we have $last(locPath.P) = T$. If T is P's actual address, then P can be registered at T.

If P moves from T before the update of $locPath.P$ has been completed, then $update.P$ will become $\langle\,\rangle$ and $last(locPath.P)$ will become T, even though T is no longer the actual address of P. While clearly a database that consistently cannot keep up with the location of the portables is not useful in practice, we would like our algorithm to be able to handle transient situations in which this happens. In our algorithm, the transient situation in which a portable moves before it has been registered at its current actual address will not violate the safety (invariant) properties of the algorithm, and the update algorithm can always catch up if at any time the portable stops moving long enough.

There are two situations that now must be reconsidered. One is the action taken when the portable moves from R to T when a token already exists. In this case, which can be recognized by checking to see whether P is registered in R, no new token should be created, but the fact that the portable has moved to T should be reflected in the state of R. This is done with a variable at R, denoted $R.Pending.P$, that contains the timestamp and destination of the latest move from R made while P was not registered at R. $R.Pending.P$ contains either a (timestamp, node) pair, whose components are denoted $R.Pending.P.ts$ and $R.Pending.P.dest$ respectively, or the value ϵ. For example, suppose P moves from R to T at a time $ts.P = k$ when it is not registered at R. Instead of a new update token being created, (k, T) is stored in $R.Pending.P$. When a token with final destination R and timestamp t arrives, if $R.Pending.P$ is not empty and if $t < k$, (k, T) becomes the new timestamp and final destination for the update token and $R.Pending.P$ is set to ϵ. The update token is forwarded toward its new final destination, removing or adding edges as before.

It is, of course, possible for the portable to move to additional destinations before the update token arrives, so we define a pending queue $Q.P$, constructed by taking the contents of $T.Pending.P$, where T is the final destination of the current token, followed by the contents of $(T.Pending.P.dest).Pending.P$, and so on. The final element of $Q.P$ is $AA.P$. $Q.P$ is $\langle\,\rangle$ if there is no *UPDATE* token for P.

Since it is possible for a portable to move away from a node and then later return to it, the contents of $T.Pending.P$ may be overwritten, invalidating the contents of some pending variables. While this is useful in practice—it eliminates cycles from Q and reduces the time needed to update the database—it also introduces the requirement that we distinguish relevant values of $T.Pending.P$ from old, invalid ones. This is the purpose of the timestamp. We require that the timestamps associated with $Q.P$ be increasing, and that all of them be smaller than $ts.P$. Then a $T.Pending.P$ entry with a smaller timestamp than an arriving *UPDATE* token for P can be recognized as old and ignored. $Q.P$ is formally defined as follows:

$$Q.P \stackrel{\text{def}}{=} \qquad (21)$$

$$\begin{array}{ll} \langle\,\rangle & \mathit{if}\ \neg token(P,\_,\_,\mathit{UPDATE},\_) \\ QQ.P.(t,T) & \mathit{if}\ token(P,\_,\_,\mathit{UPDATE},(t,T)) \end{array}$$

where

$$QQ.P.(t,T) \stackrel{\text{def}}{=} \qquad (22)$$

$$\begin{array}{ll} \langle\,\rangle & \mathit{if}\ T.Pending.P = \epsilon\ \vee \\ & \quad T.Pending.P = (tt,TT) \wedge tt < t \\ T :: QQ.P.(tt,TT) & \mathit{if}\ T.Pending.P = (tt,TT) \wedge tt > t \end{array}$$

(Observe that we need not consider a case where $tt = t$; this is a consequence of (1).)

$Q.P$ and the pending variables satisfy invariants that the timestamp of the portable is always larger than the timestamp for any pending update

$$(\forall T :: T.Pending.P \neq \epsilon : T.Pending.P.ts < ts.P) \qquad (23)$$

and, from the definition of Q, timestamps corresponding to the elements of Q are in increasing order and larger than the timestamp of the UPDATE token.

One final invariant provides the essence of the reason the algorithm is correct:

$$last(locPath.P +\!+\ update.P +\!+\ Q.P) = AA.P \qquad (24)$$

Finally, we can add an optional optimization. To motivate its introduction, suppose that at some point $AA.P = T$, and then P moves elsewhere and later returns to T (so that $AA.P = T$ again). If the update algorithm is very slow, then T will appear twice in $Q.P$. Clearly, all successors to T in $Q.P$ can be removed without violating (24). This can be done by setting $T.Pending.P$ to ϵ. We emphasize that this action merely performs an optimization; it could be eliminated altogether or delayed without affecting the correctness of the algorithm.

4.4 Algorithm

The database maintenance algorithm is given in figures 2, 3, and 4. Figure 2 shows the action that is performed when portable P changes its actual address from node Old to node New. Figure 3 gives the action taken when an update token from node R is received by node S. Figure 4 shows the action that performs the optional optimization.

Initial conditions are as follows (for all portables P and nodes R):

$$
\begin{aligned}
& HA.P = AA.P \ \wedge \\
& P \in (HA.P).Registered \ \wedge \\
& R.Out = \epsilon \ \wedge \\
& R.Pending.P = \epsilon \ \wedge \\
& \neg token(P, \_, \_, UPDATE, \_)
\end{aligned} \tag{25}
$$

$AA.P := New;$
if $P \in Old.Registered \rightarrow$
  ALL PREVIOUS UPDATES COMPLETED, UNREGISTER P AND LAUNCH A NEW UPDATE TOKEN
  $Old.Registered := Old.Registered - P;$
  $send(P, Old, nextNode(Old, New), UPDATE, (k, New));$
  if $nextNode(Old, New) \neq nextNode(Old, HA.P) \rightarrow$
    THE PATH FROM $HA.P$ WILL BE EXTENDED, ADD AN OUTGOING P-EDGE TO $Old.Out$
    $Old.Out := Old.Out \cup \{(P, nextNode(Old, New))\}$
  [] $nextNode(Old, New) = nextNode(Old, HA.P) \rightarrow$
    skip
  fi
[] $P \notin Old.Registered \rightarrow$
  AN UPDATE OF P'S LOCATION IS ALREADY IN PROGRESS, SAVE NEW LOCATION AND TIMESTAMP
  $Old.Pending.P := (k, New)$
fi

Figure 2: Node Old is notified that P has moved to New at a point where $ts.P = k$.

```
if  T ≠ S →
    Token has not yet reached final destination, forward along update path
    send(P, S, nextNode(S,T), UPDATE, (t,T));
    if  nextNode(S,T) = nextNode(S, HA.P) →
          Token moving toward home address, remove edge
          S.Out := S.Out − (P, R)
    []  nextNode(S,T) ≠ nextNode(S, HA.P) →
          Token moving away from home address
          Add new edge in direction of token, maintain single outgoing P-edge
          S.Out := (S.Out − (P, R)) ∪ {(P, nextNode(S,T))}
    fi
[]  T = S →
    Token at its final destination
    if  S.Pending.P ≠ ε ∧ S.Pending.P.ts > t →
          There are still pending updates, generate update token to next update
          send(P, S, nextNode(S, S.Pending.P.dest), UPDATE, S.Pending.P);
          if  nextNode(S, S.Pending.P.dest) = nextNode(S, HA.P) →
                  Token moving toward home address, remove edge
                  S.Out := S.Out − (P, R)
          []  nextNode(S, S.Pending.P.dest) ≠ nextNode(S, HA.P) →
                Token moving away from home address
                Add new edge following token and maintain single outgoing P-edge
                S.Out := (S.Out − (P, R)) ∪ {(P, nextNode(S, S.Pending.P.dest))}
          fi;
          S.Pending.P := ε
    []  S.Pending.P = ε ∨ S.Pending.P.ts < t →
                No more updates, register P
                S.Registered := S.Registered ∪ {P};
                S.Out := S.Out − (P, R);
                S.Pending.P := ε
    fi
fi
```

Figure 3: S receives token $(P, R, S, \mathit{UPDATE}, (t, T))$ from R.

```
if  New = AA.P →
    New.Pending.P := ε
fi
```

Figure 4: New clears $Pending.P$.

4.5 Proof

The conjunction of the invariants that make up the formal specification ((7), (11), (12), (13), and (14)) and those of section 4.4 can be proved in a straightforward way by checking that it (the conjunction of invariants) holds initially and is preserved by each action. The details are not illustrative and are omitted. Certain properties of acyclic graphs that are needed are given in appendix D.

With this done, what remains is to prove progress property (8). Update tokens not yet at the destination indicated in the last field are sent to the next node in the path to that destination. This doesn't change until the token reaches the destination. Also, $Q.P$ does not increase unless the portable's actual address changes. Thus we have the following easily-checked property:

$$\begin{aligned}
&(token(P, R, S, \mathit{UPDATE}, (t, T)) \wedge |Q.P| = k \wedge AA.P = R \wedge S \neq T) \;\; \mathit{next} \\
&\quad (token(P, R, S, \mathit{UPDATE}, (t, T)) \wedge |Q.P| \leq k \wedge AA.P = R) \vee \\
&\quad (token(P, S, nextNode(S, T), \mathit{UPDATE}, (t, T)) \wedge |Q.P| \leq k \wedge AA.P = R) \vee \\
&\quad AA.P \neq R
\end{aligned} \tag{26}$$

Applying the token progress theorem (59), we get

$$\begin{aligned}
&(token(P, _, S, \mathit{UPDATE}, (t, T)) \wedge |Q.P| = k \wedge AA.P = R) \;\; \leadsto \\
&\quad (token(P, _, T, \mathit{UPDATE}, (t, T)) \wedge |Q.P| \leq k \wedge AA.P = R) \vee \\
&\quad AA.P \neq R
\end{aligned} \tag{27}$$

Once the update token is at its destination, then either the portable will be registered at its destination, or a new token will be launched and the length of $Q.P$ decreased, or the portable will move. This is formally indicated by the following easily-checked property:

$$\begin{aligned}
&(token(P, S, T, \mathit{UPDATE}, (t, T)) \wedge |Q.P| = k \wedge AA.P = R) \;\; \mathit{next} \\
&\quad (token(P, S, T, \mathit{UPDATE}, (t, T)) \wedge |Q.P| \leq k \wedge AA.P = R) \vee \\
&\quad (token(P, _, _, \mathit{UPDATE}, _) \wedge |Q.P| < k \wedge AA.P = R) \vee \\
&\quad P \in R.Registered \vee \\
&\quad AA.P \neq R
\end{aligned} \tag{28}$$

Since all tokens must be received (5), from the PSP theorem[4] we get:

$$\begin{aligned}
&(token(P, S, T, \mathit{UPDATE}, (t, T)) \wedge |Q.P| = k \wedge AA.P = R) \;\; \leadsto \\
&\quad (token(P, _, _, \mathit{UPDATE}, _) \wedge |Q.P| < k \wedge AA.P = R) \vee \\
&\quad P \in R.Registered \vee \\
&\quad AA.P \neq R
\end{aligned} \tag{29}$$

[4] See Appendix A for this and other axioms and theorems about leads-to used in this proof.

Applying disjunction on S, we then get:

$$
\begin{aligned}
&(token(P, \_, T, UPDATE, (t, T)) \wedge |Q.P| = k \wedge AA.P = R) \quad \leadsto \qquad (30)\\
&\quad (token(P, \_, \_, UPDATE, \_) \wedge |Q.P| < k \wedge AA.P = R) \vee \\
&\quad P \in R.Registered \vee \\
&\quad AA.P \neq R
\end{aligned}
$$

Combining the above progress properties and using disjunction on T and induction (on $|Q.P|$, which clearly is always non-negative) we get:

$$
\begin{aligned}
(token(P, \_, \_, UPDATE, \_) \wedge AA.P = R) \quad \leadsto \qquad (31)\\
P \in R.Registered \ \vee \ AA.P \neq R
\end{aligned}
$$

From invariant (16),

$$\neg token(P, \_, \_, UPDATE, \_) \ \wedge \ AA.P = R \ \Rightarrow \ P \in R.Registered \qquad (32)$$

and hence

$$
\begin{aligned}
\neg token(P, \_, \_, UPDATE, \_) \ \wedge \ AA.P = R \ \leadsto \qquad (33)\\
P \in R.Registered \vee AA.P \neq R
\end{aligned}
$$

Applying disjunction to (31) and (33), we get the desired result:

$$AA.P = R \quad \leadsto \quad P \in R.Registered \vee AA.P \neq R \qquad (34)$$

5 The database query algorithm

5.1 Overview

A query to determine the location of a portable P is initiated by sending a query token. The query token travels along the acyclic graph toward P's home address until it finds a node where P is registered or where there is an outgoing P-edge. Outgoing P-edges are followed until a node where P is registered is found. When the address where P is registered is determined, a result token is generated and sent back to the originator. Since we assume that routing between all nodes in the fixed network is available, the result token is sent directly back to the originator. This is the only token type in the algorithms presented here that is not restricted to traveling only along edges in the acyclic graph.

A query token has the structure

$$(P, N, M, QUERY, Orig) \qquad (35)$$

where P is the ID of the token to be located, $QUERY$ is the message type, and $Orig$ is the ID of the node to which the result should be sent.

A result token has the structure

$$(P, T, Orig, RESULT) \qquad (36)$$

where T is the registered address of P.

5.2 Specification

Ideally, we would like to require that it always be the case that the T field of a result token contains the current registered address of the portable. This is too strong for two reasons. First, it will take some time for the reply token to get back to the originator of the lookup. Since we can't constrain the movement of the portable, the most that can be done is to say that the reply token contains the registered address of the portable at the instant the token was sent. Also, if the portable moves faster than the database maintenance algorithm can keep up with, it is not possible to expect that a correct location for the portable can be returned. What we specify is that if at some point $P \in T.Registered$ and there is a query token for P in the system, then eventually a reply token indicating that P is registered at T will be generated, or P will move. Since this must be true for all T, if P stays in one place "long enough", then a token with its current registered address will be generated. The above requirement can be specified with a leads-to property as follows:

$$\begin{aligned} P \in T.Registered \wedge token(P, \_, \_, QUERY, R) \quad &\leadsto \\ token(P, T, R, RESULT) \vee P \notin T.Registered& \end{aligned} \tag{37}$$

We also require that a query token remain in existence until a result token is generated, and that the result token contain (at the moment it is generated) the registered address. Therefore a query token that is generated when an update is in progress will eventually be responded to once the database update algorithm has caught up with the token position for long enough[5].

$$\begin{aligned} &token(P, \_, \_, QUERY, R) \quad next \\ &\quad token(P, \_, \_, QUERY, R) \vee \\ &\quad (\exists T :: token(P, \_, R, RESULT) \wedge P \in T.Registered) \end{aligned} \tag{38}$$

5.3 Algorithm

Figure 5 gives the action required to generate a query token. Note that we have not given an action for the case where the portable is registered at the node wishing to locate the token; a local lookup suffices in this case. Figure 6 describes processing on receipt of a query token.

[5] Wang [9] simply assumed that this would not happen, and did not give sufficient details to determine what would happen with his algorithm if it did.

Initial conditions are as specified in (25), plus the following (for all portables P):

$$\neg token(P, \_, \_, QUERY, \_) \ \wedge \qquad (39)$$
$$\neg token(P, \_, \_, RESULT)$$

---

if $R.Out.P = \epsilon \rightarrow$
  No outgoing P-edge, create query token and send toward P's home address
  $send(P, R, nextNode(R, HA.P), QUERY, R)$
[] $R.Out.P \neq \epsilon \rightarrow$
  Create query token and send it along P's location path
  $send(P, R, R.Out.P, QUERY, R)$
fi

Figure 5: Node R initiates query for portable P.

---

if $P \in T.Registered \rightarrow$
  P found, create result token
  $send(P, T, R, RESULT)$
[] $P \notin T.Registered \rightarrow$
  if $T.Out.P = \epsilon \rightarrow$
    No outgoing P-edge, forward query token toward P's home address
    $send(P, T, nextNode(T, HA.P), QUERY, R)$
  [] $T.Out.P \neq \epsilon \rightarrow$
    Forward query token along P's location path
    $send(P, T, T.Out.P, QUERY, R)$
  fi
fi

Figure 6: Node T receives token $(P, S, T, QUERY, R)$.

5.4 Proof

We sketch the proof of progress property (37). First, we would like to use the token progress theorem (59) to conclude that as long as the registered address remains constant long enough, the query token will eventually arrive at the registered address. We need the following:

$$\begin{aligned} &(token(P, R, S, QUERY, R) \wedge P \in T.Registered \wedge S \neq T) \quad next \\ &\quad (token(P, R, S, QUERY, R) \wedge P \in T.Registered) \ \vee \\ &\quad (token(P, S, nextNode(S, T), QUERY, R) \wedge P \in T.Registered) \ \vee \\ &\quad P \notin T.Registered \end{aligned} \tag{40}$$

This can be shown from the program text provided the following are invariant:

$$\begin{aligned} &P \in T.Registered \wedge S \neq T \wedge S.Out.P = \epsilon \quad \Rightarrow \\ &\quad nextNode(S, HA.P) = nextNode(S, T) \end{aligned} \tag{41}$$

and

$$\begin{aligned} &P \in T.Registered \wedge S \neq T \wedge S.Out.P \neq \epsilon \quad \Rightarrow \\ &\quad S.Out.P = nextNode(S, T) \end{aligned} \tag{42}$$

These follow from invariants (12) and (13) of the database maintenance algorithm (which state that if P is registered at T, then the database edges labeled P correspond to the shortest path from $HA.P$ to T) and properties of acyclic graphs. From the token progress theorem (59) and disjunction[6] over S, we get:

$$\begin{aligned} &token(P, \_, \_, QUERY, R) \wedge P \in T.Registered \quad \leadsto \\ &\quad token(P, \_, T, QUERY, R) \vee P \notin T.Registered \end{aligned} \tag{43}$$

Now, we need it to be true that if the query token is at the registered address, either a result token will be generated eventually, or the portable will move. From the program text, we obtain

$$\begin{aligned} &(token(P, S, T, QUERY, R) \wedge P \in T.Registered) \quad next \\ &\quad (token(P, S, T, QUERY, R) \wedge P \in T.Registered) \ \vee \\ &\quad (token(P, T, R, RESULT) \wedge P \in T.Registered \ \vee \\ &\quad P \notin T.Registered \end{aligned} \tag{44}$$

We combine this with (5) and the PSP theorem to get

$$\begin{aligned} &token(P, S, T, QUERY, R) \wedge P \in T.Registered \quad \leadsto \\ &\quad (token(P, T, R, RESULT) \wedge P \in T.Registered) \ \vee \\ &\quad P \notin T.Registered \end{aligned} \tag{45}$$

[6] See Appendix A for this and other axioms and theorems about leads-to used in this proof.

We then apply disjunction on S to get:

$$\begin{aligned} & token(P, \_, T, QUERY, R) \wedge P \in T.Registered \quad \leadsto \\ & \quad (token(P, T, R, RESULT) \wedge P \in T.Registered) \quad \vee \\ & \quad P \notin T.Registered \end{aligned} \tag{46}$$

We can then combine this with (43) (using the cancellation theorem) to get:

$$\begin{aligned} & token(P, \_, \_, QUERY, R) \wedge P \in T.Registered \quad \leadsto \\ & \quad token(P, T, R, RESULT) \vee P \notin T.Registered \end{aligned} \tag{47}$$

Finally, we look at safety property (38). This holds by inspection, provided the action on receiving a query token is well defined. In particular, we need the following to be invariant:

$$P \notin T.Registered \wedge T.Out.P = \epsilon \quad \Rightarrow \quad T \neq HA.P \tag{48}$$

This guarantees that $nextNode(T, HA.P) \neq \epsilon$ and thus that sending the token from T to $nextNode(T, HA.P)$ is well defined. The invariant follows from the invariants given for the database maintenance algorithm; the proof is omitted.

6 Discussion

We have given an algorithm for location update in mobile networks. Our work was inspired by Wang's algorithm [9]. Although the hierarchical approach he presents has much potential for scaling up to large systems, we found his presentation difficult to understand. A major reason was that he included details that are indeed important, but orthogonal to the design of the location database itself and best considered separately. We avoided unnecessary detail by careful abstraction in the problem formulation. For example, we do not specify the relationship between the identification numbers assigned to portables. Instead, we specify only what is necessary: that there is a connected acyclic graph imposed on the nodes of the network and that, given the ID of a portable, its home address node can be determined. While this could be decoded from the ID itself, as in Wang's algorithm, other alternatives can also be imagined. For example, one might imagine that in the future a portable ID would be associated with a person for a lifetime, rather than with the geographical location of the current home address, and the home address would be looked up in a database. Our approach allows this problem to be solved independently from the problem of finding the current address of the portable given its home address, and greatly simplifies the presentation of our algorithm.

Another important abstraction we made was viewing our "intelligent network" as an arbitrary connected acyclic graph in which any node in the graph could potentially be the actual address of a portable. This effectively decouples the logical structure from the structure of the physical network, with several significant advantages. One is that we can treat all portables, whether out of their service areas, switched off, or active in their service areas, in a uniform way simply by introducing a virtual node associated with each home address node to record the status of a portable that is switched off or out of its service area. This simplifies the algorithm and is easily implemented. Also, individual base stations need not be the optimal granularity for location information for portables. For example, several base stations might be combined in a region, with connection established with individual portables by paging in the entire region. [3] discusses the tradeoff between paging and the granularity of location information that is maintained. Finally, our approach allows fault tolerance to be introduced using well-known techniques for replication. This problem can be addressed independently of our algorithm, another example of "separation of concerns".

Finally, we specified, developed, and verified our proposed algorithm using a systematic, well-founded method and described the algorithm using a well-defined notation [1, 4, 5] with clear assumptions. This makes clear what the algorithm is and what problems we have and haven't actually solved. For example, one potential problem area for any algorithm keeping track of the current location of portables is what happens if the portable moves too fast for the database to keep up with. Wang was not clear about what happens in this situation in his algorithm, and does not indicate how connection requests (database queries, in our algorithm) are handled during a database update. Our claims for our algorithm are much more specific, and we believe our algorithm to be significantly more robust than his. Regardless of how fast the portable moves, all of the safety (invariant and next) properties will be satisfied. In particular, a portable will never have an incorrect registered address. We also know that the database algorithm will eventually catch up with a portable that stops moving. We cannot guarantee, however, that the portable's actual address will ever be registered if the portable perpetually moves faster than the algorithm can keep up with. Determining whether this is likely to happen in practice requires a performance analysis with knowledge of the expected behavior of portables and the speed of communication and computation in the fixed network. Such analysis is better carried out separately from reasoning about the correctness of an algorithm.

Acknowledgments

The authors thank Peter Hofstee for his careful reading and valuable comments, and Mani Chandy and Flemming Andersen for helpful discussions.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

[2] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–91, February 1991.

[3] J. G. Markoulidakis and E. D. Sykas. Method for efficient location area planning in mobile telecommunications. *Electronics Letters*, 29(25):2165–2166, December 1993.

[4] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[5] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[6] S. Mohan and R. Jain. Two user location strategies for personal communications services. *IEEE Personal Communications*, pages 42–50, 1994.

[7] E. D. Sykas and M. E. Theologou. Numbering and addressing in IBCN for mobile communications. *Proceedings of the IEEE*, 79(2):230–241, February 1991.

[8] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, 3rd edition, 1996.

[9] J. Z. Wang. A fully distributed location registration strategy for universal personal communication systems. *IEEE Journal on Selected Areas in Communication*, 11(6):850–860, June 1993.

A UNITY

In this section we give brief informal definitions of some key terms and state major axioms and theorems. Refer to [1] or [4, 5] for more complete information.

A.1 Definitions

Leads-to properties. $P \leadsto Q$ means that if at some point P holds, then eventually Q will hold.

Invariants. I is an invariant if it holds initially and is preserved by every action. If $I \wedge J$ is invariant, then so is I.

Next properties. P *next* Q means that whenever P holds, Q holds after execution of the next action. Since the next action may be one that doesn't change any variables mentioned in P (for example, an action that occurs on some other node), this introduces the requirement that $[P \Rightarrow Q]$.

A.2 Axioms

Basis for leads-to.

$$\frac{p \; ensures \; q}{p \leadsto q} \tag{49}$$

(p *ensures* q means that if $p \wedge \neg q$ holds at some point in the computation, then it continues to hold until q holds, and further, if p holds, then there is some action that establishes q.)

Transitivity of leads-to.

$$\leadsto \text{is transitive} \tag{50}$$

Disjunction for leads-to. If S is a set of predicates:

$$\frac{< \forall p : p \in S : p \leadsto q >}{< \exists p : p \in S : p > \leadsto q} \tag{51}$$

A.3 Theorems

PSP (Progress-Safety-Progress).

$$\frac{p \leadsto q, \quad r \; next \; s}{p \wedge r \leadsto (q \wedge s) \vee (\neg r \wedge s)} \tag{52}$$

Induction. If M is a total function from program states to W, where $(W, <)$ is well-founded (for example, less-than over the positive integers), and m ranges over W:

$$\frac{< \forall m :: p \wedge (M = m) \leadsto (p \wedge (M < m)) \vee q >}{p \leadsto q} \tag{53}$$

Cancellation.

$$\frac{p \leadsto q \vee r, \quad r \leadsto s}{p \leadsto q \vee s} \tag{54}$$

B Communication channels and tokens

Formally, we model a unidirectional (logical) communication channel between nodes r and s as a bag called ch_{rs}. In the algorithm at hand, channels are bags that model communication channels that reliably deliver all messages sent but may reorder them. In cases in which channels deliver messages in the order they were sent, channels may be modeled as sequences. Formally, sending a message corresponds to adding a message to a channel:

$$\begin{aligned} send(P, R, S, Data) \;&\stackrel{\text{def}}{=} \\ ch_{RS} &:= ch_{RS} \cup (P, R, S, Data) \end{aligned} \tag{55}$$

Receiving a message $(P, R, S, Data)$ can only occur when the message is in the channel and has the effect of removing the message from the channel.

$$
\begin{aligned}
receive(P, R, S, Data) \quad & \stackrel{\text{def}}{=} \\
& \textit{if} \;\; (P, R, S, Data) \in ch_{RS} \rightarrow ch_{RS} := ch_{RS} - (P, R, S, Data) \\
& [] \;\; (P, R, S, Data) \notin ch_{RS} \rightarrow skip \\
& \textit{fi}
\end{aligned} \tag{56}
$$

The predicate *token* is defined formally as

$$token(P, R, S, Data) \quad \stackrel{\text{def}}{\equiv} \quad (P, R, S, Data) \in ch_{RS} \tag{57}$$

The following property, introduced in section 3.3 as (5), was sufficient to prove all the other progress properties in the paper:

$$token(P, R, S, Type, Data) \quad \leadsto \quad \neg token(P, R, S, Type, Data) \tag{58}$$

But we need to know when this property is satisfied by an implementation. If all messages sent are eventually delivered (which is a reasonable assumption about the underlying system), and the algorithm has a command to handle every message that might come through and all tokens are unique (both of which we need to check), then we can conclude that (5) holds.

Note that if there can be more than one token with fields $(P, R, S, Type, Data)$ in the channel between R and S at the same time, then we cannot guarantee that there will ever be a point at which no token with fields $(P, R, S, Type, Data)$ is in the channel. Thus, we need uniqueness. If tokens are not actually unique, we can make them so by adding an auxiliary (unimplemented) data field containing unique sequence numbers.

C Token progress theorem

Often, tokens are sent along some path in a network until a node is reached where some condition holds. The token progress theorem is helpful in showing progress in algorithms that use tokens. We assume that there is a connected acyclic graph structure on the nodes of the system and that the functions $minPath$ and $nextNode$ are as defined in section 3.2.

Theorem. Suppose that the following safety property is satisfied:

$$
\begin{aligned}
& (token(P, R, S, data) \wedge S \neq T \wedge p \wedge q) \quad next \\
& \quad (token(P, R, S, data) \wedge p \wedge q) \quad \vee \\
& \quad (token(P, S, nextNode(S, T), data) \wedge p \wedge q) \quad \vee \\
& \quad \neg q
\end{aligned} \tag{59}
$$

Then

$$(token(P, \_, S, data) \wedge p \wedge q) \quad \leadsto \quad (token(P, \_, T, data) \wedge p) \quad \vee \neg q \tag{60}$$

Proof.

$$\begin{array}{ll}
& (59) \\
\Rightarrow & \quad \{ \text{ PSP theorem with (5) } \} \\
& token(P, R, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (token(P, S, nextNode(S, T), data) \wedge p) \quad \vee \neg q \\
\Rightarrow & \quad \{ \text{ disjunction on } R \ \} \\
& token(P, _, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (token(P, S, nextNode(S, T), data) \wedge p) \quad \vee \neg q \\
\Rightarrow & \quad \{ \text{ strengthen left side } \} \\
& |minPath(S, T)| = k \wedge token(P, _, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (token(P, S, nextNode(S, T), data) \wedge p) \quad \vee \neg q \\
\Rightarrow & \quad \{ \text{ property of connected acyclic graphs: } \} \\
& \quad \{ \ |minPath(S, T)| = k \wedge k > 0 \Rightarrow \} \\
& \quad \{ \qquad |minPath(nextNode(S, T), T)| < k \} \\
& |minPath(S, T)| = k \wedge token(P, _, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (|minPath(nextNode(S, T), T)| < k \wedge \\
& \qquad\quad token(P, S, nextNode(S, T), data) \wedge p) \quad \vee \neg q \\
\equiv & \quad \{ \text{ let } l = |minPath(S, T)| \text{ if } token(P, _, S, data) \ \} \\
& l = k \wedge token(P, _, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (l < k \wedge token(P, _, nextNode(S, T), data) \wedge p) \quad \vee \neg q \\
\Rightarrow & \quad \{ \text{ induction on } k \ \} \\
& l = k \wedge token(P, _, S, data) \wedge S \neq T \wedge p \wedge q \quad \leadsto \\
& \qquad (token(P, _, T, data) \wedge p) \quad \vee \neg q \\
\Rightarrow & \quad \{ \text{ disjunction with } \} \\
& \quad \{ \ token(P, _, T, data) \wedge p \wedge q \quad \leadsto \quad token(P, _, T, data) \wedge p \ \} \\
& token(P, _, _, data) \wedge p \wedge q \quad \leadsto \quad (token(P, _, T, data) \wedge p) \vee \neg q \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
& (60)
\end{array}$$

D Properties of graphs

The proofs of our algorithms rely on the following properties of undirected connected acyclic graphs:

For any two nodes R and S, there is a unique acyclic path between R and S. This allows us to conclude that if we have produced some acyclic path between R and S, that path must be the shortest path between R and S. (61)

$$S :: minPath(S,T) = reverse.(T :: minPath(T,S)) \tag{62}$$

$$\begin{aligned} S = nextNode(R,T) \ \wedge \ (nextNode(S,T) = nextNode(S,W)) \ \Rightarrow \\ nextNode(R,T) = nextNode(R,W) \end{aligned} \tag{63}$$

$$\begin{aligned} nextNode(S,T) \neq nextNode(S,R) \ \equiv \\ minPath(S,T) \text{ is disjoint from } minPath(S,R) \end{aligned} \tag{64}$$