Integration of Location Services in the Open Distributed Office

Mike Rizzo, Peter F. Linington, and Ian A. Utting
Computing Laboratory, University of Kent
Canterbury, Kent CT2 7NF
United Kingdom

Keywords: distributed systems, location systems, active badges, ubiquitous computing.

Abstract

There has recently been much interest in location systems which enable people and equipment to be tracked as they move within and across buildings. Thus far, such systems have been used in isolation with the result that, although there are often several sources of location information available at any one site, users have to consult each system individually. Additionally, it is difficult for services requiring location information to take advantage of all these sources.

In this paper, we put forward the notion of a *master location system* to co-ordinate the location process so that available sources of information are used automatically in as efficient a manner as possible. We discuss a number of factors that are of concern to the design of such a system, and describe a particular implementation which we worked on as part of an office automation project.

1 Introduction

There has recently been much interest in location systems which enable people and equipment to be tracked as they move within and across buildings. Perhaps one of the more innovative of these is the ORL active badge location system [WHFG92], where tracking is done by means of IR communication between badges and a network of sensors. Potential applications include replacing telephone numbers with personal codes, recording a history of events to aid one's memory [McC94], teleporting user interace environments [BRH94], and tracking a moving object through a video camera network.

There are often several sources of location information available at any one site. Examples include system login information, and personal diary systems. Additionally, people tend to have known habits that others may become accustomed to. For example, Mike often forgets to wear his active badge, but normally works in his room (S25) and goes for a coffee break at 15:30. Or a particular OHP projector may be kept in one specific room most of the time, but is sometimes borrowed for use in the adjacent room.

Of course, it is possible that not every object will be traceable by any one particular location system within an organisation. For example, one person may choose not to wear a badge, and the tea trolley is unlikely to log in to the computer system!

In the light of the above, we propose that a location system should not rely solely on one specific mechanism, but should try to use and combine information from as many sources as possible. It would be useful then, to have some kind of master location system (MLS) which co-ordinates all location systems available within an organisation. In this model, all such location systems are treated as slaves and are invoked by the MLS as it deems necessary. Applications requiring location information are clients of the MLS and do not communicate with the slaves directly.

It is also worth pointing out that the notion of an MLS is also useful as an abstraction for federating locators across sites. A person from site A may query the location of a person at another site B without needing to know anything about the specifics of location mechanisms employed by site B. Related to this is the issue of control over information release; individuals, or indeed organisations, may wish to restrict access to their location systems.

In this paper we discuss the issues that need to be taken into consideration in the design of such a location system. We also describe the location system developed as part of the Open Distributed Office (ODO) project [RLU94].

2 Factors influencing the design of an MLS

The availability of multiple, independent slave locators can be extremely useful, but requires a carefully designed mechanism to harness their potential effectively. This section identifies the issues that must be taken into consideration by such a design.

2.1 Uncertainty

There may be a degree of uncertainty associated with a returned location. For example, a user may be known to be logged on a particular host from a particular terminal, but is reported to have been idle for 2 minutes. Depending on the user's habits, there may or may not be a very good chance that he or she is still sitting at the terminal. With active badges, a person may have his back temporarily turned to the sensor so that the badge is not picked up for a short time. Thus a last sighting time of 2 minutes could mean that the person may still actually be in the room, though obviously with less likelihood than for a last sighting time of a few seconds.

We stipulate therefore, that whenever a locator suggests a location, it should also supply a *confidence value* to give some indication of the likelihood of that location being correct. The master locator can then use this value in its decision-making process as will be made clearer in the coming sections.

2.2 Returning multiple locations

A locator may be capable of returning more than one location for the same target object. For example, a person may have overlapping appointments in a personal diary, or may be logged in from more than one terminal on the local network. In such circumstances, the confidence value described in the previous section would normally (but not necessarily) be different for each location. For example, if a user is logged in on host A with an idle time of 3 minutes, and on host B with an idle time of a few seconds, then one would expect the confidence value associated with host B to be at least as strong as host A. In fact a typical 'confidence function' would indicate a much stronger likelihood for host B in this case. Ultimately, however, an appropriate confidence function must be based on the habits of each individual user or entity, and therefore it is not desirable, nor is it necessary, to impose constraints on the choice of such functions.

The master locator itself is capable of returning multiple locations. Clients of the master locator would be expected to try locations in order of decreasing likelihood.

2.3 Corroboration and conflict

A location indicated by one slave locator may conflict with or corroborate that returned by another. In the case of corroboration, it might be useful to combine the confidence values in some appropriate way that reflected the strength of the combined support for that location.

As is the case with locators returning multiple conflicting locations, confidence values are needed to determine which location should be tried first whenever two slave locators give conflicting results. However, there is a subtle difference in that the confidence values are now being computed by *different* locators. The respective confidence functions must therefore be chosen carefully so as to ensure that it makes sense to compare values returned by different locators. Failure to do so will result in situations where one locator is 'over-confident' relative to another.

2.4 What is a location?

If a locator (be it master or slave) is to return a location, then some kind of representation for locations is needed. This is not a trivial problem as there is often more than one way of identifying a location in space. For example, three different ways of identifying the multimedia lab at UKC are:

- room `SW103`,
- the room with telephone extension `3813`,
- the room where the host `rowan.ukc.ac.uk` is situated.

A slave locator may very well be capable of reporting one or more of these. In particular, the master locator should attempt to return as much knowledge about the location as possible, as this increases the options available to the client. For example, in the ODO project, the client may contact a person via the phone or via a workstation's audio hardware.

The problem of location comparison is also an imporant issue. If we are to establish whether one returned location corroborates or conflicts with another, then some means of effectively comparing locations is needed. In the above example, the master locator for UKC should be able to establish that room SW103 is the same as the room with extension 3813, which is the same as the room that houses `rowan.ukc.ac.uk`.

2.5 Efficiency and Strategy

In many circumstances it is highly desirable that the location system should respond as quickly as possible. A typical example where performance is an issue is the setting up of phone calls. There is no point in keeping the caller waiting for all slaves to respond if at least one of them has already established the location of the target with a high enough level of confidence.

Thus a strategy is needed to determine the sequencing of slave locator requests. A good strategy would take the following factors into consideration:

- known characteristics and habits of the target person: if the person is known not to wear a badge, then there is little point in trying the badge slave locator first. On the other hand, the badge locator would probably be a good first choice for those known to wear one regularly;
- confidence level: if the confidence level for a returned location is below some threshold, then it would probably be worth consulting some other slave locator first. This may result in corroboration of the first location, or might return some other location, possibly with a higher confidence value;
- failed trials: assuming that the MLS only returns one location at a time, if the MLS returns a location which the client finds to be incorrect, then subsequent invocations by the client should not receive that location again, even if another slave corroborates it;
- slave locator delay characteristics: some slave locators may take longer to respond than others. The availability of expected response times for slave locators is useful in the formulation of efficient location policies.

2.6 Location policy

Many of the factors discussed so far will be influenced by policies defined at both the organisation and individual level. The confidence function for badges may well be defined at the organisation level, whereas location strategy depends very much on the characteristics of the target entity. While organisation policy could be hard-wired into the master locator or supplied via a designated interface, a mechanism is needed whereby individuals can specify and alter their own policies. A good organisation policy would give individuals a lot of flexibility in areas where their habits tend to differ.

An individual's policy must be interpreted in the context of the constraints imposed by the organisation's policy. For example, if an organisation possesses a slave locator which it knows to be highly reliable and efficient in locating all locatable entities within the organisation, then it can impose that this slave locator be tried first, regardless of the policies supplied for those entities.

2.7 Extensibility

By ensuring that all slave locators present identical interfaces, a 'plug-and-play' location system can be built. This will allow slaves to be added and withdrawn dynamically. If the master locator also has the same interface, then it can itself be used as a slave to a higher-level master, thereby facilitating federation of location systems. One case where this might be useful is when a remote site's master locator needs to be consulted as part of a local location request. For example, an individual may specify that, as a last resort, an attempt should be made to locate him at some other site where he will be with some other specified person.

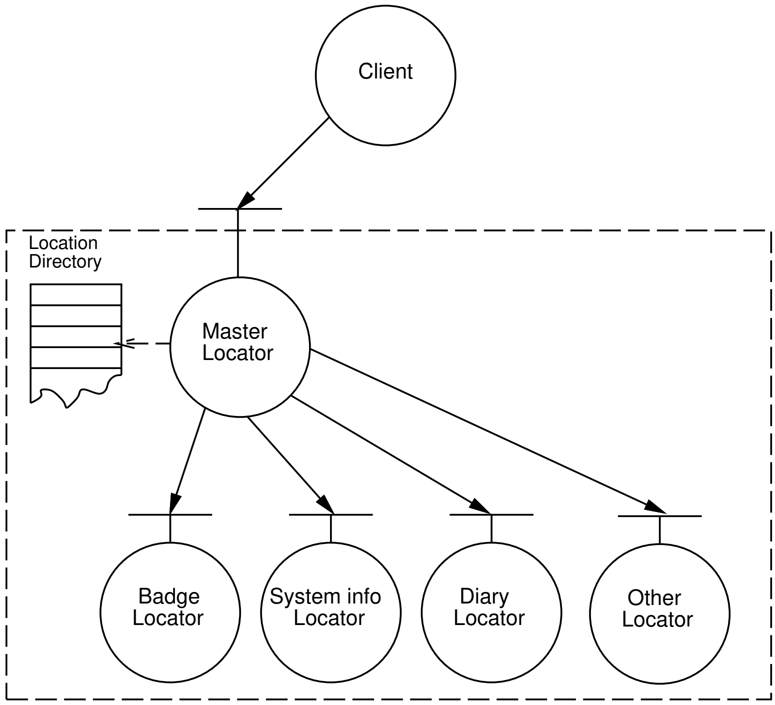

Figure 1: The ODO Location System

3 The ODO Location System

The ODO location system was designed taking all the above factors into consideration. Figure 1 illustrates the relationship between master and slave locators and introduces a new component, namely the *location directory*, which is used to store information needed for the location process.

3.1 Location representation

A location is represented as a list of name-value pairs (known as an *attribute list* in ODO). There are standard names for attributes such as room number, extension number and resident user terminal[1]. Where necessary a value may in fact consist of a list e.g. there may be more than one phone in the room.

Two location representations are taken to refer to the same location if (i) they have at least one attribute name in commmon and, (ii) for every name in common, the values are either equal, or in the case of list values, share at least ONE common list member. If (i) is not satisfied then nothing can be inferred about the equivalence of the two locations. If (i) is satisfied but (ii) is not, then the locations are taken to be distinct[2].

One of the roles of the location directory is as an aid to comparing location representations. This directory contains full location representations for all locations at a site.

[1] Examples of user terminals include workstation consoles and X terminals.

[2] The underlying assumption here is that any location representation returned by a locator represents a unique location.

Whenever a location is returned by a slave, the master can expand this representation using data from the directory so that condition (i) is always satisfied when comparing locations.

Expanding representations in this way serves to maximise the options available to the client. For example, if a slave returns a location in terms of a terminal name, then an expanded representation will probably also contain a room number and an extension number.

3.2 Location policy

The location directory also serves as a repository for individuals' location policies. A directory entry corresponding to an individual will contain attributes which collectively specify that individual's location policy. Where no policy is specified by an individual, a default organisation policy is assumed.

A directory entry pertaining to an individual may also contain data that is needed by one or more of the slaves. For example, a capability[3] to read an individual's diary may be left in that individual's directory entry so that it may be passed on to the diary slave locator.

At the organisation level, policy can be hard-wired into the master locator. For example, the corroboration function used to calculate a new confidence level when two slave locators agree on a location is established uniquely in the master locator.

3.3 Access control

Control over access to location services is also by the use of capabilities. Each locator requires presentation of a valid capability before it can start to provide service. Access control is therefore achieved by limiting the availability of such capabilities.

In ODO, capabilities are stored as attributes in directory entries. Users have their own personal directories where they can store information, including capabilities, pertaining to other users and services. Capabilities may also be made available in shared or public directories. Placing a capability in a directory effectively grants that capability to any user who has access to that directory. Access to directories is, in turn, also controlled by capabilities. This approach is extremely flexible, making it possible to limit access to designated individuals, groups, and/or organisations.

Turning back to the location directory, whilst maintenance of directory entries for locations is best left to the system administrator, maintenance of directory entries for individuals is best left to the individuals themsleves. Individuals are given sufficently powerful capabilities corresponding to their respective location directory entries, allowing them to change their location policies and other attributes that may influence the location process.

3.4 Location session

In order for a client to use the location system it must first establish a *location session* via the MLS. This is necessary because the MLS needs to maintain state pertaining to the client's request, and because the client may need to communicate with the MLS several times before it obtains the information that it needs (or gives up). For example, the MLS might first suggest one location which the client finds to be incorrect. Subsequently, the

[3] Capabilities are used extensively for access control in ODO. For an introduction to capability-based protection see [Mul89].

client asks the MLS whether it can come up with another location. In order to do this, the MLS must have knowledge of the prior invocation and take the associated result into account i.e. it must remember the first incorrect location and must ensure that it is not proposed to the client again.

For similar reasons, the MLS must itself initiate a session with each of the slave locators that it uses. Like the MLS, slave locators are also capable of proposing one more than one location, and the MLS may need to interact with them several times. This similarity is useful because it fits in nicely with the idea of using the same interface for both the MLS and its slaves.

3.5 Locator interface

In accordance with the desire for a 'plug-and-play' environment, the ODO location system uses the same interface for all locators, be they slave or master. Figure 2 shows the ANSAware IDL [Arc] specification for the `Locator` interface.

The `Locate` operation receives a locator capability (a client can only invoke a locator if it has a capability for it) and a specification of the target individual (in ODO, individuals are also represented by attribute lists). If successful, this operation opens a new location session and returns:

- the location perceived most likely to be correct by the locator;
- an integer confidence level between 0 and 100, where 0 indicates that the locator has no confidence in the returned location at all (but does not know it to be impossible) and 100 indicates absolute certainty;
- a session voucher which can be used to carry on with the location process if the returned location turns out to be wrong.

In the event that a client finds the returned location to be incorrect, it can present the session voucher to `NextLocation` in order that it may obtain the next most likely location. If the list of all possible locations is exhausted, the locator returns `L_ListExhausted` and closes the location session.

If a client finds a returned location to be correct, it should terminate the location session by calling `Release`.

Finally the `GetInfo` operation can be used to return the current status of a session.

3.6 Master locator algorithm

In short, the algorithm works by placing possible locations on a priority queue on the basis of confidence level. When the topmost entry has a confidence level above a session threshold, this entry is returned. After all slaves have been exhausted, entries from the queue are returned until success is reported or the queue is exhausted.

The remainder of this section describes the algorithm in greater detail using C++-like pseudo-code. For simplicity, the description of the state and pseudo-code assume a single session. In practice, multiple instances of the state and corresponding multiple threads of execution can exist simultaneously, and references to state should therefore be interpreted as being relative to a session. Another simplification is that details pertaining to locator capabilities are omitted.

```
Locator : INTERFACE =
NEEDS Types;
BEGIN

    L_OpReason : TYPE = {
        L_LocatorFailure, L_ListExhausted, L_InvalidVoucher
    };

    L_LocateResult : TYPE = CHOICE OpStatus OF {
        OpSuccess => RECORD [
            voucher : odo_Datum,  -- location voucher
            loc : odo_DirEntry,
            conf : CARDINAL
        ],
        OpFailure => L_OpReason
    };

    Locate : OPERATION [
        sc : odo_SCapability;
        who : odo_DirEntry
    ] RETURNS [ L_LocateResult ];

    NextLocation : OPERATION [
        voucher : odo_Datum
    ] RETURNS [ L_LocateResult ];

    Release : OPERATION [
        voucher : odo_Datum
    ] RETURNS [ ];

    L_GetInfoResult : TYPE = CHOICE OpStatus OF {
        OpSuccess => RECORD [
            who : odo_DirEntry,
            loc : odo_DirEntry,
            conf : CARDINAL
        ],
        OpFailure => L_OpReason
    };

    GetInfo : OPERATION [
        voucher : odo_Datum
    ] RETURNS [ L_GetInfoResult ];

END.
```

Figure 2: IDL specification for Locator interface

```
exhausted = FALSE;
loc_queue.clear();
tried_list.clear();
cur_slave = loc_policy.list.first();
voucher = NULL;
```

Figure 3: Session initialisation

Session state

For each session, the following state variables are maintained:

- a target specification `target`, which is an attribute list that describes the entity to be located;
- a location policy `loc_policy`, which consists of two fields, namely `list`, a list of references to slave locators, and `threshold`, a threshold confidence value;
- a priority queue `loc_queue`, which is used to store the results of invocations on slave locators. Every element in the queue is a `LocationEntry` structure containing the fields `loc` for the location, and `conf` for the confidence level. The queue is ordered by decreasing confidence level;
- a list `tried_list`, that records all locations returned by the master locator as the session progresses;
- a slave reference `cur_slave`, that keeps track of the current slave, and a session voucher `voucher`, which stores the voucher that needs to be presented to the current slave in order to obtain the next location. The latter is initialised to `NULL` each time `cur_slave` is updated. Subsequently, values are assigned to it from the results of invocations to `Locate` and `NextLocation` on the current slave locator interface;
- a flag `exhausted`, which is used to indicate that the policy's list of slaves has been exhausted.

For all lists (including `loc_queue`, which is described in terms of a list), it is assumed that the following operations exist: `empty()` returns `TRUE` if the list is empty, `FALSE` otherwise; `contains(i)` returns `TRUE` if the list contains `i`, `FALSE` otherwise; `first()` initialises an internal counter to point to the first item in the list and returns this item or NULL if the list is empty; `next()` advances the internal counter and returns the current item or NULL if the end of the list has been reached; `clear()` removes all items in the list; `append(i)` adds the item `i` to the end of the list; `insert_before(i,j)` inserts the item `j` before item `i` in the list; `remove(i)` removes the item `i` from the list.

The label `NULL` is used to denote null (or zero) values and can assume any type as required by the context in which it is used.

Session behaviour

A master location session is automatically created when the `Locate` operation is invoked on the master locator interface. Assuming the location policy has been obtained (using the

```
done = FALSE;
while (!exhausted && !done) {
  if (voucher == NULL) {                     // No current session
    if (cur_slave != NULL) {                 // Haven't reached end of the policy
      rloc = cur_slave.Locate(target);       // Call Locate on current slave
    }
    else {                                   // No more slaves listed in policy
      exhausted = TRUE;
    }
  }
  else {                                     // Session already open
    rloc = cur_slave.NextLocation(voucher); // Call NextLocation on cur slave
  }

  if (!exhausted) {                          // If we have a result
    if (rloc denotes success) {              // containing a loc
      add_queue(rloc.loc,rloc.conf);         // add the loc to loc queue
      voucher = rloc.voucher;                // update session voucher
    }
    else {                                   // This slave is no longer useful
      voucher = NULL;                        // Reset session voucher
      cur_slave = loc_policy.list.next();    // Advance cur_slave
    }
  }

  // We've found next likely loc if the loc queue has something in it, and
  // one loc's confidence value is above the threshold
  done = (
    !loc_queue.empty() &&
    (loc_queue.first().conf >= loc_policy.threshold)
  );
}

if (!loc_queue.empty()) {                    // If we have a loc to return
  LocationEntry le = loc_queue.first();      // extract it
  loc_queue.remove(le);                      // remove it from loc queue
  tried_list.append(le.loc);                 // append it to tried list
  succeed(le);                               // return this loc
}
else {
  fail(L_ListExhausted);                     // Fail with reason L_ListExhausted
}
```

Figure 4: Obtaining the next location

```
add_queue(Location loc, int conf) {
  Expand loc using location directory

  if (!tried_loc.contains(loc)) {  // If loc not returned already
    LocationEntry i;

    // Scan loc_queue, searching for an occurence of the same location
    for ( i = loc_queue.first(); i != NULL ; i = loc_queue.next()) {
      if (loc and i.loc describe the same place) {
        conf = corroborate(conf, i.conf);
        locqueue.remove(i);
        break;
      }
    }

    // Create a new location entry for the new location
    LocationEntry *le = new LocationEntry(loc, conf);

    // Scan loc_queue, searching for appropriate place to insert le
    for (i = loc_queue.first(); i != NULL ; i = loc_queue.next()) {
       if (i.conf <= conf) {
         break;
       }
    }

    // Insert le before first entry with a lower confidence value
    loc_queue.insert_before(i, *le);
  }
}
```

Figure 5: The procedure `add_queue`.

supplied target entity specification and the master locator directory, or by copying the default location policy if the target entity has not provided a location policy) and is stored in `loc_policy`, figure 3 illustrates the steps needed to initialise a master locator session.

Once the session has been initialised, the `Locate` operation returns the first most likely location using the routine described in figure 4. This routine is also used to obtain the next most likely location in the case of the `NextLocation` operation. The routine makes use of an additional local flag `done` to indicate that the next most likely location has been determined, and a variable `rloc` to receive the results of `Locate` and `NextLocation` invocations.

`add_queue()` (figure 5) inserts a location entry at the right place in the priority queue, but only if the location has not been tried already. If the location is already present in the queue, then its confidence level and position are updated if necessary. The new confidence level is calculated by applying a corroboration function. Note that this algorithm makes it impossible for the same location to appear in the queue more than once.

Corroboration function: corroborate()

The corroboration function is determined by the organisation policy. Some possible choices are described here, highlighting a couple of important issues in the process.

Given two confidence levels $l1$ and $l2$, the corroboration function defined by

$$l = max(l1, l2)$$

represents a 'quality rules over quantity' policy. For example, if two slave locators both return confidence levels of 60% for a location A, and another returns a confidence level of 61% for location B, then the master locator returns B before A because the confidence level for B is higher than the combined levels for A, which is 60%.

A better function would increase a confidence level by some appropriate amount each time a locator expressed some confidence in the corresponding location. One possibility is:

$$\begin{aligned} &lmax = max(l1, l2) \\ &lmin = min(l1, l2) \\ &l = lmax + trunc((lmin/100) * (100 - lmax)) \end{aligned}$$

With this function, two 60%'s for A would win over one 61% for B because the corroboration function yields a confidence level of 84% for A. However, because this function relies on `min()` and `max()`, it is sensitive to the order of arrival of confidence values.

Finally, the function defined by:

$$l = 100 - (100 - l1) * (100 - l2)/100$$

preserves the intention of the previous one but remains insensitive to arrival order[4].

4 Tuning and performance indicators

The framework encompasses several tuning points which can be used to make the system more effective. Suitable choices for confidence functions, thresholds, corroboration functions, and location policies can only be made on the basis of experience with a working system. However, the framework can provide some level of support by providing performance indicators.

Such indicators can be compiled by the master locator by having it record a log for every location session. Each log is effectively a indication of how successful the client found the location results to be. Each time a client asks for the next possible location, this can be interpreted to mean that the previously returned location was found to be incorrect. On the other hand, an invocation of `Release` can be taken to indicate that the client found the last returned location to be correct. If, over a period of time, the total number of failures exceeds some chosen threshold then this indicates that some parameters may require adjustment.

Detailed logging, such as recording the identity of slave locators and the confidence values they return, coupled with a comprehensive statistics reporting system, facilitates the

[4]The reasoning behind this function is as follows: let p_A denote the probability that the location returned by locator A is correct, let p_B denote the probability that the location returned by locator B is correct. Then the probability that both A and B are wrong is $(1 - p_A)(1 - p_B)$. We are interested in the probability that this does not happen i.e. that it is not the case that both A and B get it wrong. This probability is given by $1 - (1 - p_A)(1 - p_B)$.

task of pin-pointing the parameters that need adjusting. For example, if the performance of the system as a whole is satisfactory, but is extremely poor for one particular entity, then a parameter pertaining to that specific entity (such as its location policy) may need adjusting. Or if it can be determined that one particular slave is not being very effective, regardless of the entity being located, then its confidence function might need to be changed.

Is should be noted that 'optimum performance' can mean different things to different users. One possible definition of optimum performance might relate to the number of slave locators used in a any one session, wherein the smaller the number of slaves used, the better the performance is considered to be. Another definition might relate to the time taken to locate an individual during office hours. It is therefore not possible to guage the performance implications of an individual's policy without a reference context. A more sophisticated master locator might allow individuals to specify how performance is to be measured for their particular requirements so that it can warn them when their policy does not live up to their expectations.

5 Discussion

Location systems are becoming of increasing importance in office automation [Hop94] and universal personal telecommunications (UPT) [Lau94] services. They are also useful in shared media-spaces such as Xerox's RAVE [G$^+$92], virtual reality systems, and memory-aid systems.

There are often several potential sources of location and different approaches to location may be required for different locatable entities, depending on their habits and characteristics.

In this paper we have described a framework for the integration of location services which allows multiple location mechanisms to co-exist and co-operate. The framework also gives the locatable entities some control over the approach used to locate them, so that each may choose an effective strategy to match its own mobility characteristics. We have also described how the framework provides support for performace tuning via indicators that report on the success rate of the locator, both on an individual entity basis and on a system-wide basis.

We have built a prototype of the ODO location system, using ANSAware [Arc92] as an engineering platform, and using the ORL active badge system and computer system login information as slave locators. This has been useful to demonstrate the principles behind our framework. We are using the prototype in order to determine what kinds of functions, values and policies are likely to be most useful.

Acknowledgments

This work was funded by EPSRC grant GR/G57864. Mike Rizzo is currently sponsored by the University of Malta. We are grateful to Olivetti Research Limited for supplying us with an active badge system.

References

[Arc] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom (apm@ansa.co.uk). *ANSAware 4.1 Application Programmer's Manual.*

[Arc92] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom (apm@ansa.co.uk). *RM.099.02: An Overview of ANSAware 4.1*, May 1992.

[BRH94] Frazer Bennett, Tristan Richardson, and Andy Harter. Teleporting—making applications mobile. In *Proceedings of 1994 Workshop on Mobile Computing Systems and Applications, Santa Cruz*, December 1994.

[G+92] William Gaver et al. Realizing a video environment: EuroPARC's RAVE system. In *Proceedings of CHI '92, Monterey, California.* ACM SIGCHI, ACM Press, New York, May 1992.

[Hop94] Andy Hopper. Communications at the desktop. *Computer Networks and ISDN Systems*, 26:1253–1265, 1994.

[Lau94] Gregory S. Lauer. IN architectures for implementing universal personal telecommunications. *IEEE Network*, 8(2):6–16, March 1994.

[McC94] John McCrone. Don't forget your memory aide. *New Scientist*, pages 32–36, February 1994.

[Mul89] S.J. Mullender. Protection. In Sape Mullender, editor, *Distributed Systems 1st ed*, chapter 7. Addison-Wesley, 1989.

[RLU94] Mike Rizzo, Peter F. Linington, and Ian A. Utting. The ODO project: a case study in integration of multimedia services. Technical Report 12-94, Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, United Kingdom, August 1994.

[WHFG92] Roy Want, Andy Hopper, Veronica Falcao, and Johnathon Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.