

Peer Data Structures

Ulrik P. Schultz
Center for Pervasive Computing
University of Aarhus, Denmark

Monday 8th of April

Abstract

In this position I present peer datastructures as a means to representing and synchronizing data in applications for pervasive computing systems. This communication abstraction is presented in the context of a virtual machine for pervasive computing applications, also outlined in this paper.

1 Introduction

Contemporary research in services for pervasive computing and ubiquitous computing often assumes a central server [2, 3, 7, 9, 11], even though from an architectural point of view the central server is ill-suited to these new computing paradigms. Examples of such centralized services includes context and location awareness [2], a platform for groupware applications [11], a location-based information system [3], as well the simpler yet still difficult problem of managing appointments in a group-based calendar.¹ The origin of this dichotomy is the simplicity of programming clients that operate through a central server, as opposed to decentralized peers that share relevant information.

Implementing a pervasive computing system without a central server typically involves defining a communication protocol for distributing and synchronizing data. Rather than implementing such protocols from scratch for every application, my goal is to have a library of standard abstractions for sharing and synchronizing data, similarly to the collection library available for any decent object-oriented language. Distributed Asynchronous Collections are a prime source of inspiration for this work [6], but are based on the publish/subscribe paradigm, which is not necessarily ideal for distributing all kinds of data. Rather than relying on a continued dissemination of data, I want each local copy of the data structure to continuously be as complete as possible. (Of course, the publish/subscribe paradigm is powerful enough to provide this behavior as well, but I am searching for other potentially simpler solutions.)

This position paper describes a work-in-progress in the implementation of communication abstractions for a pervasive computing virtual machine: I describe the current state of our virtual machine implementation (including its basic communication mechanisms), outline how to implement peer data structures as an abstraction for distributed communication with simple communication and synchronization behavior, and last give examples of how these data structures concretely can be used as a substitute for a centralized server architecture.

¹To be fair, the purpose of the cited papers was to demonstrate the usefulness of services, not to investigate the architecture of pervasive computing systems.

Motivating examples

As a first example, consider a distributed calendar. Here, an appointment is made for an entity (e.g., a person or a room) for a specific time interval on a specific day. A conflict between two appointments usually needs to be resolved by the end user. The standard approach is to store the calendar information on a central server, but if this server is down, only locally cached information can be read, and no new appointments can be made. Alternatively, the calendar information can be replicated between servers, but a broken network connection between e.g. a meeting room where appointments are being made and the servers is sufficient for rendering the system inoperable.

As a second example, consider a location service used to implement a context service: the complete set of location information is used in an expert system that infers information about what each object (e.g., person) is doing currently, to for example allow the graphical user interface to be context-sensitive [2]. The location information need not be precise, since context information by its nature is based on guessing. A central server is typically used to collect location information which is then accessed by clients. If the server or the network connection is down, no information can be gathered, not even about local events.

2 Virtual machines in pervasive computing

In the context of this paper, I define a pervasive computing system as a distributed system with heterogeneous devices and network connections of variable quality. Devices are not only heterogeneous in the sense of having different processors, but also range from e.g. low-end embedded systems through cell-phones and PDAs to distributed computing clusters and large server systems. Network connections range from high-quality local-area network through wide-area networking as embodied by the Internet to unstable network connections provided by wireless networks or Bluetooth.

To facilitate application development for pervasive computing systems, we are at the Center for Pervasive Computing currently developing a virtual machine with special facilities for supporting pervasive computing, named SOM. Due to the heterogeneous nature of pervasive computing systems, virtual machines have an obvious advantage since they provide a uniform execution environment. The SOM virtual machine executes programs written in a minimal version of Smalltalk. A single virtual machine hosts many independent, self-contained programs, referred to as *systems*. A system can migrate between virtual machines (and hence physical machines); we support strong migration, so program execution continues immediately after the point where the migration primitive was invoked.

We intend for applications to be structured in terms of multiple systems, to facilitate the use of agent-oriented programming, and also e.g. allow the central parts of an application or the GUI component to migrate physically close to the user. Thus, communication between systems should be as simple as communicating with local objects; proxy objects are ideal for this purpose.

3 Proxy objects

In SOM, communication between systems can be both local and remote, depending on the current physical location of each system. We have chosen to make communication as transparent as possible by using proxy objects, as described in this section. Note that the proxy mechanism is currently under implementation, and not all aspects are completely functional.

The primary inter-system communication mechanism of SOM is remote method invocation through proxy objects. A proxy is created by contacting a remote system and looking up an object stored in its global dictionary. Objects never move between systems, and passing objects as arguments to methods results in proxy objects being passed; the migration of complete systems is the only way to move data, although objects can be copied across systems, as described in the next paragraph. Proxies should be transparent with regards to system migration, which is implicitly the case since they are based on SOM TCP/IP connections which are stable with regards to system migration (an alternative approach would be the way object migration is implemented in Emerald [10]).

Proxies support a special primitive `clone`, which creates a copy of the remote object in the local system. The local clone references objects from its original system using proxies (being Smalltalk, this includes the class). An object can override the default clone operation to e.g. provide deep cloning.

Proxies are implemented at the virtual machine level, which allows the virtual machine to aggressively optimize communication between systems that have migrated to the same virtual machine. Our goal is to optimize proxy calls similarly to virtual dispatches, and even inline code across system boundaries. Deoptimization can be used to recover the system boundaries when a system needs to migrate to a different virtual machine. Thus, system mobility becomes an important factor in communication, since a system can migrate closer to other systems with which it communicates frequently. In addition to optimizing local communication, optimization across physical machine boundaries can be done for objects that permit a latency in when modifications become visible to their proxies, since object state can be cached locally and considered immutable for the duration of the latency interval. (Object immutability comes as a special case when there is an infinite latency interval.)

4 Peer data structures

Just as the individual parts of a standard application needs to share data across module boundaries, a pervasive computing application needs to share data across physical devices (as an example, consider the calendar and location service described in the introduction). In this section I first argue for peer data structures as a solution to this problem, and then give two concrete examples of peer data structures, a dictionary and a disjoint set.

4.1 The case for peer data structures

Pervasive computing applications need to share part of their state with other devices in the pervasive computing system: critical state must be replicated to other devices to permit autonomous behavior without a network connection, but not all state can be replicated since the devices are of widely different capacity. I propose to store shared data in a *peer data structure* where the complete data structure is copied across all interested parties and incrementally synchronized between these parties. (Naturally, synchronization policies are application specific and must be defined by the programmer, similarly to e.g. comparison operators for ordered sets.)

A peer data structure is created on a system, and can be replicated to other systems; both the original and any replicate are valid representatives for the data structure and are henceforth referred to simply as *peers*. Data can be read from and written to the data structure using standard operations. Providing consistency for complex updates (e.g., transactions) would probably be hard (if not impossible

with this approach), but compound elements in the data structure can be used to achieve a similar effect.

Resolution of synchronization conflicts for a data structure is defined by the programmer, although default behavior can be provided (examples follow in Section 4.2). A given data structure generates synchronization conflicts in specific cases, e.g. a peer dictionary generates a conflict only for separate insertions of items with identical keys. The programmer specifies a per-element merge operation that merges a conflicting modification into the data structure, possibly signaling an error to the originator of the conflicting modification. Merge operations are required to give rise to the desired semantics of the data structure independently of the order in which modifications arrive, so that a semantically correct state results from a series of modifications that arrive in random order. Thus, groups of peers that are disconnected from the rest of the system and that make local updates will, when reconnected, resynchronize their local updates with all other updates, until a stable configuration is reached.

Synchronization is done incrementally from peer to peer, similar to gossiping [4, 5]. Specifically, synchronization of a modification only requires synchronization with the immediate neighboring peers before the process is completed locally. Synchronization with all remaining peers continues asynchronously, and can thus generate synchronization errors at any future point in time, until the synchronization process is complete. Merging is done at each peer before synchronizing with other peers, which reduces traffic (similarly to message obsolescence [8]). If a neighbouring peer is unavailable within a given timeout, the unsynchronized element is tagged as “modified but not synchronized,” and will be synchronized when either the peer becomes available or a new peer is assigned instead.

By default, a peer neighbors the peer it was created from and a number of its peers, as well as any peers subsequently created from it; the neighboring relation can be modified as long as all peers are transitively connected. A system can use this rerouting mechanism in case a peer has crashed (detectable through a timeout). Maintenance of neighboring relations is considered future work, but I expect that the existing work on gossiping would come in useful here.

In effect, at any given point in time, the actual content of the peer data structure exists throughout the pervasive system, but no single device is guaranteed to have complete information. Thus, each peer represents the local “best guess” of the global state of the data structure.

4.2 Peer dictionary

A dictionary is a data structure that maps a key to an element; it supports the following operations:

Dictionary `d`:

```
d.insert(k,e)    : insert (k,e) into d, overwriting any (k,e')
d.lookup(k)      : if (k,e) is in d then return e
d.containsKey(k) : if (k,e) is in d then true, else false
```

Note that I for simplicity have omitted a `delete` operation; deletion introduces many complications, and it may be that a more restricted version is more appropriate. Thus, in a peer dictionary, synchronization conflicts can occur only when insertion is done for an existing key. Conflicts are resolved by merging the original element and the new element, as follows:

```
d.insert(k,e') with d.containsKey(k):
  e:=d.get(k); d.insert(k,merge(e,e'))
```

```

non-strict latest-arrival:
    element = data
    user_merge(e,e') = e

keep-newest (elements carry time stamp):
    element = (time stamp,data)
    user_merge(e,e'): if time(e) newer time(e') then return e
                      else return e'

conflict-preserving:
    element = NO_CONFLICT(data) | CONFLICT(id,data)
    user_merge(e,e'):
        id:=fresh_id();
        originator(e').signal_conflict(id);
        HERE.signal_conflict(id,e');
        return CONFLICT(id,e)
    user_merge(CONFLICT(id,e),RESOLVE(id,e')) = return e'

```

Figure 1: Predefined merge operators with corresponding data representations

Some elements can be automatically merged, so the user-defined merge operator is only needed in some cases:

```

merge(e,e'): if mergeable(e) then return e.merge(e')
             else return user_merge(e,e')

```

Example of elements that can be automatically merged are other peer data structures and objects that have a merging behavior explicitly defined.

Figure 1 shows three simple examples of user-defined merge operators and corresponding data representations (the most common merge operators would be available in the library of peer data structures). A realistic application would probably require much more elaborate merge operators. In each case I give the representation of an element and the definition of the `user_merge` operator. The “non-strict latest-arrival” merge operator simply keeps the latest modification, and is appropriate for information that need not be exact. The “keep-newest” merge operator uses a time stamp to actually keep newest information. The “conflict-preserving” merge operator keeps any merge conflicts around and leaves it up to the originator or the local peer (represented by “HERE”) to resolve the conflict with another modification.

4.3 Peer disjoint set

A disjoint set is a data structure that contains non-overlapping elements, with the definition of overlapping being a parameter of the data structure; based on the overlapping criterion each peer can decide individually whether an element is legible for inclusion in the data structure (modulo synchronization conflicts). This data structure supports the following operations:

```

Disjoint set s with overlap condition:
s.insert(e)   : If not s.overlap(e) then insert e into s
s.overlap(e)  : If e overlaps with element in s then true, else false
s.contains(e) : If e is in s then true, else false
s.delete(e)   : Remove e from s, if present

```

(Note that by defining the `overlap` parameter as equality, we get a standard set.) The insertion behavior is similar to that of the peer dictionary:

```

s.insert(e) with s.overlap(e):
  es:=s.get_overlaps(e); // returns all overlapping elements
  for each x in es: s.delete(x);
  for each x in es: s.insert(merge(x,e))

```

The merge operators presented for the peer dictionary (shown in Figure 1) can be used straightforwardly for the peer disjoint set.

5 Return of the central server

Returning to the central server examples of the introduction, these are easily implemented with peer data structures, as follows:

- The calendar system can be composed from peer data structures:
`Calendar = Dictionary(entity,Appointments)<no user_merge>`
`Appointments = Dictionary(date,Daily)<no user_merge>`
`Daily = ConstrainedSet<no overlapping hours,conflict preserving>`

For the two dictionaries, no `user_merge` operator is needed since merging is done automatically by the aggregated data structure. In the last case, the constrained set, no overlapping hours are allowed for appointments, and all conflicts are preserved and signaled. The user is required to resolve each conflict manually.

- The location service can simply be implemented using a dictionary that maps entities to locations. Either “non-strict latest-arrival” or “keep-newest” can be used; the latter is significantly more precise but has a larger overhead since some concept of global time is needed. (Having the complete location information available across all interested peers allows any peer to infer context information, which again can be stored in a peer data structure.)

I assume that existing applications from these examples written for a central server could easily be rewritten to use peer data structures. Of course, the behavior of the applications would be significantly modified compared to a central server solution, but it was never the intention to provide the exact same semantics.

I am currently working on prototype implementations of the distributed data structures described in this paper. In addition, I am searching for a replacement for the traditional proxy mechanism. For all of their simplicity, proxies have a major disadvantage in the context of pervasive computing: a proxy can only be used so long as the target object is accessible. If there is no network connection to the physical machine where the target object is currently stored, operations on the proxy will block. As a solution, I am currently investigating *peer objects* which replace the implicit client-server relation in proxies with a peer-to-peer relation. A peer object would be an autonomous clone of a remote object that synchronizes modifications with this object (again, synchronization conflicts are handled by user code).

Acknowledgements

The SOM virtual machine is being developed in collaboration with Kasper V. Lund, Jakob Roland and Mads Torgersen, with kind assistance from Lars Bak.

References

- [1] Gregory D. Abowd, Barry Brumitt, and Steven Shafer, editors. *Ubiquitous Computing*, Atlanta, Georgia, USA, September 2001.

- [2] Jakob E. Bardram and Henrik Bærbak Christensen. Middleware for Pervasive Healthcare - A White Paper. In Guruduth Banavar, editor, *Advanced Topic Workshop—Middleware for Mobile Computing*. <http://www.cs.arizona.edu/mmc/Program.html>, Heidelberg, Germany, November 2001.
- [3] Frederik Espinoza, Per Persson, Anna Sandin, Hanna Nyström, Elenor Caccitore, and Markus Bylund. *GeoNotes: social and navigational aspects of location-based information systems*. In Abowd et al. [1], pages 2–17.
- [4] P. Eugster, A.-M. Kerrmærc R. Guerraoui, S. Handurukande, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the IEEE Symposium on Dependable Systems and Networks (DSN 2001)*. 2001.
- [5] Patrick Th. Eugster and Rachid Guerraoui. Probabilistic multicast. To appear at the 3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002), June 2002.
- [6] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 252–276, Cannes, France, 2000. Springer-Verlag.
- [7] Joseph F. McCarthy and Theodore D. Anagnost. EVENTMANAGER: Support for the peripheral awareness of events. In Thomas and Gellersen [12], pages 227–235.
- [8] José Pereira, Luís Rodrigues, and Rui Oliveira. Semantically reliable multicast protocols. In *IEEE Intl. Symp. on Reliable Distributed Systems (SRDS'2000)*. October 2000.
- [9] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: a service framework for ubiquitous computing environments. In Abowd et al. [1], pages 56–75.
- [10] R.K. Raj, E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, January 1991.
- [11] Jörg Roth and Claus Unger. Using handheld devices in synchronous collaborative scenarios. In Thomas and Gellersen [12], pages 187–199.
- [12] Peter Thomas and Hans W. Gellersen, editors. *Handhelds and Ubiquitous Computing*, Bristol, UK, September 2000.