Programming with Hierarchical Maps

Peter Ørbæk
Department of Computer Science,
University of Aarhus,
Aabogade 34,
8200 Aarhus N, Denmark
poe@daimi.au.dk

Abstract

This report describes the hierarchical maps used as a central data structure in the Corundum framework. We describe its most prominent features, argue for its usefulness and briefly describe some of the software prototypes implemented using the technology.

1 Introduction

Palpable computing aims to take pervasive computing to the next level in terms of usefulness and understandability for end users. Six challenges shape the notion of palpable computing: invisibility complemented with visibility, scalability complemented with understandability, construction complemented with de-construction, heterogeneity complemented with coherence, change complemented with stability, and sense-making complemented with user-control.

This report describes the hierarchical maps of the Corundum [Ørb05] framework. Inspired by the Plan 9 OS from Bell Labs [PPD+95a, PPD+95b] and the Linux `/proc` file system, the Corundum framework uses a hierarchical map to support the implementation of palpable components, services and devices. In particular, the use of the externally accessible hierarchical map supports construction/deconstruction of assemblies of services, and visibility and introspection of services and applications.

The rest of this report is structured as follows: In Section 2 we introduce the most prominent features of the hierarchical maps and argue for their usefulness, in Section 3 we give some examples of their use in programming. Section 4

describes the access permission system of the maps in more detail, and in Section 5 we describe how message handlers are integrated. Section 6 details the C++ interface to the maps. Section 7 presents related work, and Section 8 briefly describes some of the software prototypes that use the maps and the Corundum framework as their platform. Section 9 analyzes the hierarchical maps in terms of their support for the palpable qualities, and Section 10 concludes and outlines directions of future work.

2 The Hierarchical Map

A hierarchical map (henceforth called an *h-map*) is a simple tree-structured name space that we use to hold (most of) the non-transient data of a single process. The well-known abstract data type can be described by the context free rules below:

```
HMap ::= TreeNode
TreeNode ::= TreeLeaf | TreeDir
TreeLeaf ::= Name Value
TreeDir ::= TreeNode*
```

Names are simple strings. Values are integers of various sizes, floating point numbers, strings, BLOBs (Binary Large Objects for images and the like), addresses, messages, message handler objects, and code pointers etc. Values are tagged with a type identifier. Nodes in the tree structure can be identified by the *path* of names from the root to the node, written eg. “a/b/c”.

The h-map is a run-time entity that can be altered dynamically: tree-nodes can be added and deleted, values can be changed. The values are dynamically typed using a tagged or “boxed” representation.

The benefit of the data structure and the novelty of the framework described here does not come from the data structure itself, but from its use. The Corundum framework uses the h-map to store, basically, all non-transient data of a process. It is used to route messages among services and components, to keep documentation as well as configuration data, to keep lists of subscribers and discovered peers, to act as the parsed representation of XML data [BPSM98], etc.

The general motto of the framework is to try to use the h-map instead of inventing ad hoc data-structures.

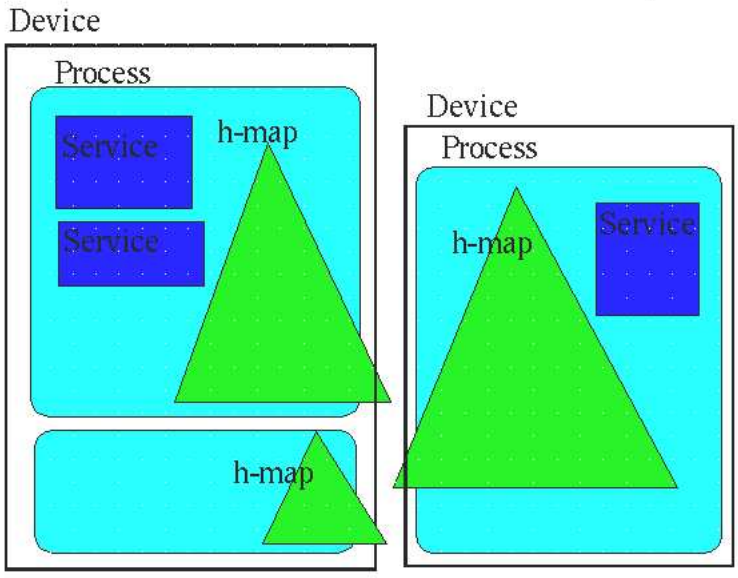

Figure 1: Two devices, one hosting two processes each with their own h-map. The h-maps extend outside the devices to illustrate that they are accessible from the outside.

The framework encourages an *extrovert programming style*, where components and services expose what they can do (potential uses, events accepted and sent), what they are doing (eg. logging), and what they have been doing (history). This is all done via the h-map which is globally visible, and accessible from outside the process over a network.

One may think of the h-map as a process-local file system where the values stored are not files but values. Alternatively, one may think of the h-map as similar to a "Document Object Model" (DOM) from the XML world, or as the *exoskeleton* of a process: it is the structure which holds together the different software parts of the process, and it is externally visible.

Processes automatically make their h-map accessible to remote processes. This facilitates a high degree of remote introspection and visibility of the structure and state of a process. No special access methods are needed to remotely access eg. the list of peer services discovered by another process, or to modify the parameters of a process, as long as they are put in the h-map of the process.

Note that we make a deliberate distinction between how one accesses the process-local h-map, and how one accesses the h-map of a remote process. The process-local h-map (shared among in-process components) is accessed via ordi-

nary synchronous method calls to, for example, read the value stored at a certain path in the local h-map. The value of a node in a remote h-map is accessed by explicitly sending an (asynchronous) message to the remote process asking for the value at a path in that h-map.

It is not possible to "mount" remote h-maps transparently in a local h-map (as is common with file systems), as this would invalidate the assumption of fast, deterministic, local access to entries in the local h-map. If mounting was allowed one would always have to access the h-map with the assumption that the access could time out because a remote device was removed or turned off. This would be much too cumbersome to use.

In short, the above distinction is made on the belief that in a fragile pervasive computing world with constantly changing connectivity, remote network communication should not be hidden as in remote method invocation (RMI) and RPC, because of the large differences in latency, reliability and failure modes between a local procedure call and a remote one.

The h-map makes remote access inside components and devices possible. It can also be used to enable unforeseen reuse of parts of components. Within the Corundum framework it is, for example, used to be able to remotely control the output of debugging information from processes, and to optionally route this information across the net at run-time, to allow on-the-fly remote logging from deployed services. This is one of the ways in which processes may expose what they are doing.

The framework encourages programmers to name their non-transient data and put it in the h-map. This allows for a more data centric model of computation, separating functionality from the data, as also advocated by [GD+04].

This, of course, works against encapsulation. But, maybe encapsulation can be considered harmful? From the programming-by-contract view encapsulation helps the provider of functionality to make the internal workings of the functionality invisible to the user, and thereby allowing a contract that is less binding for the provider. Exposing more of the internals puts a heavier burden on the provider and provides more insight and more opportunities for unforeseen reuse for the user.

However, the framework does not force programmers to put everything in the h-map, and encapsulation of the internal workings of components can be

achieved by not putting all data into the h-map. See also the section on the permission system below.

With traditional object oriented programming, using popular patterns like Facade, Layers, Proxy etc. [BMR+96] one often has to go through several interfaces, and method calls to access the data one wants, especially if functionality is being grafted onto an architecture afterward as the result of new user demands. Hence the often observed need for restructuring [BL76] and refactoring [CCD+98, Opd92] when the architecture evolves.

The h-map allows access to deeply nested data in the h-map from anywhere via a single simple interface. It is simple to use the Visitor pattern to work on a sub-tree of the h-map as well. Many common data structures map nicely onto the h-map instead of using ad hoc structures with their own idiosyncratic interfaces. (lists, vectors, sets, trees,...). This may allow for more loosely coupled systems and fewer necessary refactorings.

The flip-side of the above is that it introduces dependencies between different software parts on the structure of the h-map, in order for other parts of the code to find a thing in the h-map it must stay in a fixed or, at least, a well defined place.

2.1 XML Support

One might also externalize the internal data of a process through for example a relational data model, but a hierarchical model is simpler to implement, fits nicely with traditional classification, and matches the tree-structure of XML documents, and scene graphs of graphics and windowing systems, etc.

To better allow for parsing XML documents into the h-map there is no uniqueness constraint on names within a single directory: a single directory may contain multiple entries with the same name. This allows to represent the following XML document in the way shown below:

```
<?xml version="1.0" encoding="iso-8859-1"?>
  <Assembly>
    <Prereq>
      <Required servicename="camera"/>
      <Required servicename="display"/>
    </Prereq>
    <Connections>
```

```
      <Connection source="camera" sink="display"
        pattern="picture-taken*"/>
    </Connections>
  </Assembly>
```

The XML document is parsed into the h-map using an Expat-based XML parser (or SAX parser in the Java case), as shown in the following sub-tree of the h-map.

```
xml/
   Assembly/
      Prereq/
        Required/
           servicename: [str: 'camera']
        Required/
           servicename: [str: 'display']
      Connections/
        Connection/
           source: [str: 'camera']
           sink: [str: 'display']
           pattern: [str: 'picture-taken*']
```

Inspired by XPath [CD99] we can distinguish between the two "Required" entries with paths of the following form: "xml/Assembly/Prereq/Required[0]" and "xml/Assembly/Prereq/Required[1]". The path "xml/Assembly/Prereq/Required" is equivalent to the path with "[0]" appended.

2.2 H-map interfaces versus traditional service interfaces

Will systems necessarily become more interdependent and brittle with h-map interfaces than with standard interfaces? No. The h-map can be used to have a classical interface with no more and no less exposition of internals.

Note also that in classical CORBA/IIOP [OMG98] rogue clients can dynamically construct method invocations with illegal types and number of parameters. Hence, CORBA also needs to do run-time type checking of parameters in a CORBA interface. Normally a compiler checking the types is relied upon, but in an unsafe environment this cannot be relied on. One can also run IIOP on top of SSL (Secure Socket Layer), but the Corundum protocols could run on SSL as well.

The h-map allows, but does not enforce the publication of a wider interface. It also allows the documentation and even formal specification of interfaces to be represented *on-line* by the devices themselves in a uniform manner. Any formal or informal interface specification could be hosted by the h-map, but a specification or recommendation of such a specification is outside the scope of this report. External, off-line, interface specifications, like CORBA IDL still have their place to be used at compile time.

2.3 Evolution

Already in 1976 Belady and Lehman [BL76] identified two laws of program evolution dynamics: the *law of continuing change*, and the *law of increasing entropy*. The first law states that a program exposed to external forces (like market pressure or technology advances) will continue to change until it is judged more cost effective to freeze it and re-create it. The second law says that the structuredness of an evolving program will decay over time unless specific work is executed to maintain a structure.

The dynamic nature of the h-map allows a large degree of flexibility: the structure will need to change over time as the system evolves, in accordance with the first law. Another benefit of the exposed structure of the h-map is that since it is plainly visible, it is more likely to be kept neat, and hence counteract the increasing entropy of the second law. Note, however, that we do not see h-maps as a "silver bullet" [FPB95] for all system design and maintenance. Also, it is only the data structure of the h-map that is visible and thus invites maintenance, whereas the h-maps provide no help for keeping the underlying code neat.

3 Examples of Use

This section outlines some examples of programming use of the framework and of the h-maps in particular. Consider the following piece of C++ code:

```
class Foo : public Bar {
  public:
    int counter;

  public:
```

```
    Foo() { counter = 0; }

    void operation(int x) { ... counter += x; ... }
};
```

Here the counter is public and may be altered by other objects within the same process, if they can obtain a pointer to the containing object. With the h-maps one would instead write:

```
class Foo2 : public Bar {
  public:
    Foo() { mydir->write("foo/counter", 0); }

    void operation(int x) {
       ...
       mydir->write("foo/counter",
                    mydir->read("foo/counter").asInt()+x);
       ...
    }
};
```

where `mydir` is a pointer to a sub-tree of the h-map relevant for this object/class. This looks somewhat more cumbersome, but makes the counter accessible from without the process. Other services and browsers may inspect and alter the counter. There are two main aspects of this: one is the externalization of the counter, allowing access to it from the outside, and the other is the rooting of the data item (the counter) in a name space (the h-map), so that it may easily be accessed both within and without the process.

This also means that the code above is more fragile, one should really write something like:

```
class Foo2 : public Bar {
  public:
    Foo() { mydir->write("foo/counter", 0); }

    void operation(int x) {
      int tmp;
      if (mydir->readInt(&tmp, "foo/counter")) {
        mydir->write("foo/counter", tmp + x);
      }
      ...
```

```
    }
};
```

where we check that there *is* an integer stored at the path “foo/counter”. Some outside entity might have set it to a string, or might have erased the directory altogether. The example is a single threaded service, and there is therefore no need to lock the entry in the h-map between `readInt()` and `write()`. For multi-threaded services, the standard mutual exclusion mechanisms apply.

Another way to cope with unintended changes from third parties is to “type lock” an entry in the h-map. This will prevent the entry from being deleted and from having its value set to a different type. Then one can safely write the above example as:

```
class Foo2 : public Bar {
  public:
    Foo() {
      mydir->write("foo/counter", 0);
      mydir->resolve("foo/counter")->setTypelock(true);
    }

    void operation(int x) {
       mydir->write("foo/counter",
                    mydir->read("foo/counter").asInt()+x);
       ...
    }
};
```

Attempting to change the value to a string would result in an exception being thrown at run-time.

An elaborate permission scheme like the ones found in file systems is also implemented, so that some may only have read-permission to a node, and others have no access at all. This is elaborated in the following.

4 Permissions

Inspired by the system of permission bits used widely in file systems, we have equipped the h-map with permission bits also, to allow control over who can

change properties of the h-map. This allows more fine grained control over what external services and clients can do to the h-map on a service.

More elaborate access control and structure preservation schemes where investigated: most notably a grammar-based scheme to control the structure of a sub-tree of the h-map, making the system ensure that the sub-tree conformed to the grammar at all times. However, this was found too rigid an approach, as it would entail that whole sub-trees would have to be created outside the global h-map and inserted as a whole, instead of building them gradually inside the h-map. This would be necessary to ensure that the grammar was always adhered to. One could loosen this by selectively switch off grammar checking at times, but this was found inelegant. File-system like permissions allow a more prototypical approach where a structure can be constructed piecemeal and later write protected.

The grammar idea could still be useful for interfacing: for informing users and other systems about what can be written where in the h-map, but this should probably be implemented at a higher level. The h-maps themselves are good for representing grammar rules, and explicit messages and methods to verify that a sub-tree of the h-map conforms to such a grammar could relatively simply be made.

More general access control list schemes for protecting the contents and structure of the h-map have been considered, but found too costly in terms of run-time performance.

Concretely, we equip each node in the h-map with three sets of bits: one set for the local process, another set for operations being carried out for other members of the community[1], and a third set of bits for operations carried out for other communities. This is like the Unix permission bits where there are permissions for owner, group and others.

Each set of permission bits contains:

- *ReadName*: If set, the name of the h-map node can be read.
- *ReadValue*: If set, the value of the h-map node can be read.
- *WriteName*: If set, the name of the h-map node can be written.

[1]Corundum services are members of *communities* and network messages are encrypted with a community specifi c key.

- *WriteValue*: If set, the value of the h-map node can be written.
- *ChangeType*: If set, the type of the value stored in the h-map node can be changed (another type of value can the stored there).
- *Handle*: If set, a message handler at the h-map node can be invoked.
- *ChangePermissions*: If set, the permission bits of the h-map node can be altered.

If an operation is attempted, which is not allowed by the permission bits, an exception is thrown (and distributed to network exception subscribers, akin to linked processes in Erlang [Arm03]).

5 Message Handlers

Message routing takes place via the h-map, so that messages between services are routed to a node within the h-map of the destination service. To handle the incoming messages, the service has installed a handler function pointer at that point in its h-map, and at message delivery time the handler procedure is called with the incoming message.

In the Corundum framework, all communication among services and processes is done via asynchronous message passing. This is part of the Corundum framework and programming style, but not intrinsically tied to the h-maps: h-maps are equally well-suited for routing synchronous RPC style calls.

```
void myHandlerProc(TreeDirectory *self, const Message& msg)
{
  // called when messages arrive for "srv/myservice/handler"
}
...
// install handler in h-map
root->write("srv/myservice/handler",
            Unit("myhandler", myHandlerProc));
```

Alternatively, one can use the Functor pattern for the message handlers, so that the messages may be handled by a `handle()` method inside a C++ class:

```
struct MyHandler : public MsgHandler {
  ...
  void handle(const Message& msg)
  {
    // called when messages arrive for "srv/myservice/handler"
  }
};

MyHandler *myHandlerObj = new MyHandler;

// install handler in h-map
root->write("srv/myservice/handler", Unit(myHandlerObj));
```

If a message is addressed to a directory in the h-map, it is automatically distributed to all message handlers below that directory. This allows for easy message distribution to multiple subscribers.

Messages are distributed to receivers in the h-map by resolving the recipient path part of the message in the local h-map, and invoking the message handler stored there (if any) with the message.

6 Interface Overview

This section shows the core of the C++ class interface to the h-maps. There are three classes corresponding to the productions of the grammar given above: the abstract class `TreeNode`, `TreeDirectory` for internal nodes of the tree, and `TreeLeaf` for the leaves holding values.

The core of the abstract super class `TreeNode` is shown below with a number of utility methods elided for readability.

```
class TreeNode : public Marshalable
{
  protected:
    TreeEntryType type; // directory or leaf
    Name          name; // name of node
    TreeDirectory *parent;  // parent pointer, not marshaled

    // protected constructors: abstract superclass
    TreeNode();
    TreeNode(TreeEntryType t, const Name& s);
```

```
private:
  PermBits permissions; // permission bits, private to
                        // TreeNode to maintain control over
                        // where they can be set.

public:
  virtual ~TreeNode();

  /* Returns the TreeEntryType type of the node. */
  TreeEntryType getType() const;

  /* Permission setter and getter */
  unsigned int getPermissions() const;
  void setPermissions(unsigned int v);

  // getter and setter for the name
  Name getName() const;
  void rename(const Name& n);

  TreeDirectory *getParent() const; // Get ptr to parent node

  // the following entries are here for ease of use,
  // allows expressions like
  // "v = dir->resolve(path)->readNode()"
  virtual void setTypelock(bool x);
  virtual bool getTypelock() const;

  virtual UnitType unitType() const;
  virtual Unit readNode() const; // read value of node
  virtual void writeNode(const Unit& u); // write value of node

  // resolves a path in the tree to a node (read only)
  virtual TreeNode * resolve(const Path& path,
                             TreeDirectory **parent_ret = NULL);
  // resolves/creates a path in the tree
  virtual TreeNode * resolveCreate(const Path& path,
                                   bool crDir = false);

  // invoke the handler stored in this node
  // with msg as parameter
  virtual void invokeHandler(const Message& msg);
```

```
    // marshaling
    virtual void marshal(MarshalStream *strm) const;
    virtual void unmarshal(MarshalStream *strm);

  private:
    /* set parent pointer */
    void setParent(TreeDirectory *p);
    friend class TreeDirectory;
};
```

The core of the `TreeLeaf` class is depicted below:

```
class TreeLeaf : public TreeNode
{
  protected:
    Unit val; // the value stored in the leaf

  public:
    TreeLeaf();
    TreeLeaf(const std::string name, const Unit& v);

    /* Return the UT_ unit type of value stored here. */
    UnitType unitType() const;

    /* Return the unit value stored at this leaf.
       Requires PERM_ReadValue permissions. */
    Unit readNode() const;
    void writeNode(const Unit& u);

    void invokeHandler(const Message& msg);

    virtual void marshal(MarshalStream *strm) const;
    virtual void unmarshal(MarshalStream *strm);
};
```

The core of the `TreeDirectory` class is shown below. The `readString()` method and the `readStringDef()` method are only exemplars. Similar methods exist for the other data types that can be stored in leaves.

```
class TreeDirectory : public TreeNode
{
  public:
```

```
  typedef std::vector<TreeNode *> TreeNodeVec;
  typedef TreeNodeVec::iterator iterator;
  typedef TreeNodeVec::const_iterator const_iterator;

protected:
  TreeNodeVec entries; // the vector of pointers to sub-trees

public:
  TreeDirectory();
  TreeDirectory(const std::string& name);
  ~TreeDirectory();

  // STL iterator stuff for accessing sub-trees
  iterator begin();
  iterator end();
  size_t size() const;

  const TreeNodeVec& getEntries() const;
  std::vector<Unit> getUnitEntries() const;

  // for remote directory listings:
  std::vector<Unit> getLDUnitEntries() const;
  std::vector<Unit> getLDUnitEntriesDeep() const;

  TreeNode * resolveLocally(const Name& nm) const;
  TreeNode * resolve(const Path& path,
                     TreeDirectory **parent_return = NULL);

  bool insertLocally(TreeNode *newnode);
  bool removeLocallyByName(const Name& name);
  bool removeLocally(TreeNode *n);
  bool remove(const Path& path);
  TreeNode * resolveCreate(const Path& path,
                           bool crDir = false);

  /* Write the value val at path in the tree. Automatically
    creates the path if it does not already exist. */
  void write(const Path& path, const Unit& val);

  /* Returns a copy of the Unit stored at path. If the path
    does not exist return an UT_Undef unit. */
  Unit read(const Path& path);
```

```
    /* Read a string from a path in the tree starting from this
      directory. Returns true if the path exists and the value
      stored is a string, and sets *result to that string.
      Otherwise returns false and leaves *result unchanged. */
    bool readString(std::string *result, const Path& path);

    /* Read a string at the path path starting from this node,
      and if not found return defval */
    std::string readStringDef(const Path& path,
                              const std::string& defval);

    void invokeHandler(const Message& msg);

    static TreeNode *startUnmarshal(MarshalStream *strm);
    virtual void marshal(MarshalStream *strm) const;
    virtual void unmarshal(MarshalStream *strm);
};
```

7 Related Work

The One.World framework by Grimm et al. [GD+04] supporting pervasive applications incorporates a notion of nested environments that are not unlike the hierarchical structure and programming model described here, but there are notable differences. In One.World the nested environment is a network-global, distributed structure shared among distributed services, whereas the h-maps described here are light weight process-local and provide fast and deterministic access.

The Blackboard pattern [BMR+96] is a shared data structure used in, for example, AI systems to share data among "agents". The pattern is used to deal with changing contexts and uses, and to deal with an evolving architecture. The h-maps described here can be used as a blackboard structure among processing agents within and without a process, but it is not a network shared structure with that purpose.

The Linda coordination language and many others [CG89, RW97] provide a distributed, synchronized tuple space shared among agents, and can be seen as a distributed implementation of the Blackboard pattern. Unlike the h-maps, the tuple space is a global shared structure, and as such associated with much higher

access latency than the local h-maps.

In [DBK05] Decker et al. describe a simple file system for small ubiquitous computing devices. However, it is a *file* system with files being byte streams like in Unix, not values like in the h-map, and it is meant for local use only, not for structuring remote access and handling communication among devices. They also focus on the structure of the hierarchical name space as opposed to its use.

8 Use Experience

The Corundum framework [Ørb05] implements the h-map ideas. The framework has been used to implement a number of prototypes within the Palcom project: the *geotagger* prototype assembling a digital still camera service with a GPS service and a PDA display service, with the purpose of adding EXIF GPS meta information the pictures taken. The *sitetracker* prototype combines a webcam service, a GPS service, a digital compass service, and a display service to provide a pointing tool to show the position of a geographic site as an overlay on a live video image. The *connectivity* prototype combines several software services on a PC with a service running on a Nokia 6600 mobile phone, to provide simple, supervised, setup of an internet connection via Bluetooth and GPRS.

All the prototypes are distributed systems consisting of a number of services running on disparate computers and devices, communicating via IP over wireless ethernet and/or Bluetooth.

The Corundum framework integrates the Expat XML parser so that XML documents can be parsed into sub-trees of the h-map. XML documents can also be produced from sub-trees of the h-map. This is used as an external representation of specific assemblies both by the simple meta-assembler tool and our graphical assembler tool supporting the graphical construction of assemblies of devices and services.

XML is used also for the specification of GUI user interfaces for a couple of our prototype services. The interface is written in XML, and parsed into the h-map, and there used as a DOM or scene graph by an OpenGL based renderer. This makes the setup of simple 3D user interfaces quite simple. The h-map representation of the scene graph is used to hold colors and textures and strings for the interface, as well as coordinate transformations. A number of visitor

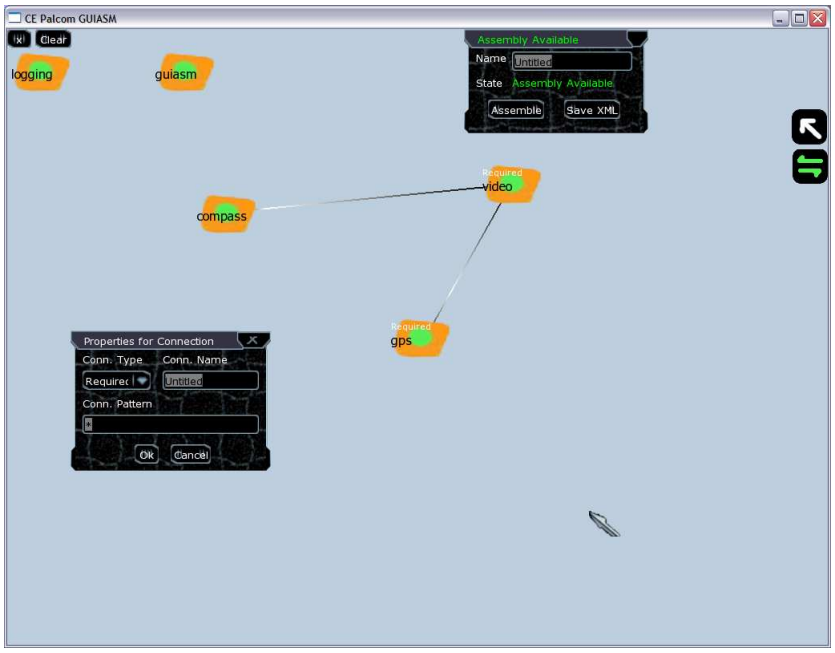

Figure 2: The graphical assembler prototype used to construct assemblies of palpable services. The assembler uses the h-map to represent the UI, and for discovery and connection of services.

classes are used to transform the scene graph and render it, as well as for input event propagation.

The h-map idea is also being implemented within the Palcom virtual machine (PRE-VM) to provide a uniform interface to VM internals and to generally support programming palpable applications and services with h-maps. An integration with the JADE agent framework [BPR00] is also being undertaken at Whitestein within the Palcom project to investigate the concepts of resilience and adaptivity in ad hoc network communication. Furthermore, the use of h-maps is being explored in connection with architectural manifestations and query languages (AQL) [IH05].

The Corundum framework, described in depth elsewhere [Ørb05], is more than the h-maps described here. The framework for example provides a multicast based service discovery mechanism, a number of bearer protocols (UDP, TCP, Bluetooth,...), a message format, and a serialization mechanism for it, encryption, point-to-point communication, multicast communication, and topic-based publish/subscribe etc.

All of these features build heavily on the h-maps: bearer protocol “drivers”

are stored in the h-map together with meta-information about the bearer; published as well as received announcement messages are kept in the h-map, and all service addressing goes via the h-map.

The framework exists in two interoperable implementations: a C++ implementation for high performance, and a Java version (JCorundum) for Java programmers. The C++ version of the core framework is implemented in about 6000 lines of well commented code, and the Java version is about the same size.

8.1 Remote H-map Access

The `Palcom::GenericNode` class (the superclass of all components and services in the framework) and its descendants handle a number of generic messages that allow remote access to the h-map of a service. H-map directories can be listed, and h-map entries can be read and written.

The generic messages also support subscriptions to events from services. Both two-way where the to-be event sink subscribes to events from a source, and three-way where an assembly connects a source to a sink. Subscriptions to remote exceptions are also handled by the generic messages. The remote exception mechanism is inspired by Erlang [Arm03].

These features are used in the prototypes to build remote inspection and manipulation tools, that allow dynamical creation and modification of *assemblies* of services.

8.2 Example Corundum H-map

The listing below is a commented dump of the h-map of an isolated instance of a simple service (`du1`), in a situation where it cannot see other services. It is one of the simplest real-world examples.

```
communities/   ←list of communities that the service is a member of
  poecomm: [int: 1]
community: [str: 'poecomm']   ←current default community
bearers/   ←bearer protocols supported
  udp: [msghandler]   ←UDP/IP unicast
  udpmcast: [msghandler]   ←UDP multicast
  tcp: [msghandler]   ←TCP/IP point-to-point
discovery/   ←things related to discovery are below here
```

```
  announcer: [msghandler]   ←handles outgoing announcements
  interval: [int: 5]   ←announcement interval in seconds
  announcement: [msghandler]   ←handler for incoming announcements
  outgoing/   ←directory containing announcements that are periodically sent from here
      logging: [message: [msg:    ←this process announces a logging service
          sender: udp:poecomm:0.0.0.0:23457:srv;logging;entry;
          recipient: udpmcast:poecomm:239.3.3.4:23456:discovery;announcement;logging()]
      du1: [message: [msg:    ←the service also announces a du1 service
          sender: tcp:poecomm:0.0.0.0:23456:srv;du1;entry;
          recipient: udpmcast:poecomm:239.3.3.4:23456:discovery;announcement;du1()]
  listeners/   ←one may install listeners here if they too need to hear incoming announcements
      diruser: [msghandler]
  received/   ←directory of received announcements, in this example we only see our own
      udp:poecomm:10.11.41.160:23457:srv;logging;entry/
        method: [str: 'logging']   ←method of the received announcement: the service type
        time: [int: 1099986779]   ←timestamp of reception
        msg: [message: [msg:    ←the received announcement message
          sender: udp:poecomm:10.11.41.160:23457:srv;logging;entry;
          recipient: udpmcast:::23456:discovery;announcement;logging([int: 20])]
        ttl: [int: 20]   ←time-to-live in seconds of this announcement, from message parameter
      tcp:poecomm:10.11.41.160:23456:srv;du1;entry/
        method: [str: 'du1']
        time: [int: 1099986779]
        msg: [message: [msg:
          sender: tcp:poecomm:10.11.41.160:23456:srv;du1;entry;
          recipient: udpmcast:::23456:discovery;announcement;du1([int: 20])]
        ttl: [int: 20]
srv/   ←services have their local configuration below /srv
  logging/
      subs/   ←subscribers for the logging service
      entry: [msghandler]   ←entrypoint for logging service
      keys/   ←logging keys control which kinds of logging messages are output
        bearer: [int: 1]   ←bearer logging messages are turned on
        node: [int: 0]   ←node logging messages are turned off
        crypto: [int: 0]
        discovery: [int: 0]
        message: [int: 0]
        xml: [int: 0]
        it0: [int: 0]
        it0mcast: [int: 0]
        it0node: [int: 0]
        asm: [int: 0]
        clnt1: [int: 1]
```

```
        to-stderr: [int: 1]   ←logging to stderr is on
        to-subscribers: [int: 1]   ←logging to logging subscribers is on
        lastKey: [str: 'clnt1']
    du1/
        subs/   ←subscribers to the du1 service
        entry: [msghandler]
xsubs/   ←subscribers to exception messages, typically assemblies.
```

9 Analysis

This section relates the h-map features to the six challenges of palpable computing. As detailed in this report externally visible h-maps aids visibility, with invisibility allowed by the permission scheme.

With respect to scalability, the h-maps scale up with the number of devices as they are local to each process. A global structure would be harder to scale up to large systems. The h-maps can be used in small as well as large devices, as the extra memory requirement of the h-maps is a small constant factor on top of the data that would otherwise have to be externalized otherwise.

One trick to keep the memory requirements of the h-map down is to keep all path components (eg. "a", "b", and "c" of the path "a/b/c") uniquely in a per-process hash table. The name of each node would then be a pointer into the unique entry for the name in the hash table, and path component comparison would be simple pointer comparison.

By making the internal structure and data visible to the outside, and to the end-user through various browser tools, the h-maps also help with respect to the understandability aspect of palpable computing.

Dynamic construction and de-construction of assemblies of service and devices is supported within Corundum, by way of the h-map that keeps event subscription lists and handles message addressing. Service discovery is also an important feature here, for which the h-map is used to keep lists of discovered services, as well as lists of periodically outgoing announcement messages.

The h-maps and our messaging protocol are platform neutral (as witnessed by the interoperable C++ and Java implementations), and thus supports heterogeneous devices, and allows making coherent assemblies of such devices.

Achieving both sense-making and user-control is not done by just using h-

map technology, and needs to be handled at a higher layer.

10 Conclusion and Future Work

We have described the idea of using an h-map as the exoskeleton of a process. The Palcom Corundum framework, building on the h-map idea, both in its C++ and Java incarnations have been used to implement a number of services to support palpable computing.

With some integration with a programming language, the use of the h-map could be less cumbersome. Programming language integration would also allow some form of type checking / inference of the types of data stored in parts of the h-map. I think that the very open-ended nature of the dynamically typed h-map should be kept, but a formalism allowing one to specify static type constraints on parts of the h-map might be useful. Possibly access control could also be integrated with a type system, so that static access constraints could be imposed as part of the types.

Scope – the set of services and sub-trees of their h-maps that can be addressed – is currently at the community level in Corundum. If two services are in the same community they can access each others h-maps in their entirety. During discussions with users of the framework it has become clear that a more fine grained scoping mechanism may be called for. It would be feasible for a service to restrict outside access to the local h-map to a sub-tree defined by the service. In this way a service within a process only makes its own sub-tree of the h-map visible.

Acknowledgments. This work has been funded by the PalCom project: an EU 6th framework programme: the Disappearing Computer in Future and Emerging Technologies (FET), part of the Information Society Technologies. Contract no: 002057.

I must also thank my colleagues in the PalCom project, especially Michael Christensen, and the student programmers Esben T. Nielsen, Jesper W. Olsen, Tony Gjerlufsen and Jacob F. Pedersen for implementing the Java version of the Corundum framework, and many of the prototypes on top of Corundum.

References

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Swedish Institute of Computer Science (SICS), Stockholm, December 2003.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, (3):225–252, 1976.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons Ltd., 1996. ISBN 0-471-95869-7.

[BPR00] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with JADE. In *Proc. of the 7th International Workshop on Intelligent Agents: Theories, Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 2000.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C, February 1998. http://www.-w3.org/TR/REC-xml.

[CCD+98] Michael Christensen, Andy Crabtree, Christian Heide Damm, Klaus Marius Hansen, et al. The M.A.D. experience. In *Proc. ECOOP'98 – Object Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 13–40, Brussels, 1998. Springer-Verlag.

[CD99] James Clark and Steve DeRose. XML path language (XPath) version 1.0. W3C Recommendation 16 November 1999, November 1999. www.w3.org/TR/xpath.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[DBK05] Christian Decker, Michael Beigl, and Albert Krohn. A file system for system programming in ubiquitous computing. In *Proc. of ARCS 2005*, volume 3432 of *Lecture Notes in Computer Science*, pages 249–264. Springer-Verlag, 2005.

[FPB95] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, twentieth anniversary edition, 1995.

[GD+04] Robert Grimm, Janet Davis, et al. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–486, November 2004.

[IH05] Mads Ingstrup and Klaus M. Hansen. A declarative approach to architectural reflection. In preparation, 2005.

[OMG98] OMG. *CORBA/IIOP 2.2: The Common Object Request Broker: Architecture and Specification*, February 1998. http://www.omg.-org/corba/c2indx.htm.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[Ørb05] Peter Ørbæk. The Corundum framework: Status March 2005. Technical Report Palcom Working Note #64, DAIMI, Aarhus, March 2005.

[PPD+95a] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Tricky, and Phil Winterbottom. Plan 9 from Bell Labs. Technical report, Bell Labs, 1995.

[PPD+95b] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Tricky, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[RW97] A. Rowstron and A. Wood. Using asynchronous tuple space access primitives (BONITA primitives) for process coordination. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and*

Models (Coordination'97), volume 1282 of *Lecture Notes in Computer Science*, pages 426–429. Springer-Verlag, 1997.