The Cell Project: Component Technology for the Internet

Ran Rinat Scott F. Smith
Department of Computer Science
The Johns Hopkins University
{rinat, scott}@cs.jhu.edu

October 11, 2001

Abstract

We propose a new component model with a focus on the needs of Internet components, mainly dynamism and security. It is based around the concept of a *cell*—a new programming abstraction with component-specific responsibilities. At the heart of it is the introduction of a *persistent link* mode of interaction into components. In existing technologies (COM/Corba/Java Beans), components interact exclusively by obtaining handles to other components on runtime and then invoking operations via interfaces. With persistent links, components are developed assuming some *imports* given; then on runtime they must link to other components which will provide the implementations for the imports, via *exports*. Cells include both models of interaction in one entity. Supported features include cross-network persistent linking, cells as containers of objects, a connection-based security mechanism, and cross-component class inheritance. These features are expected to prove especially beneficial for components in the Internet.

1 Introduction

Component-based software systems are recently receiving significant attention in both industrial and research environments. Traditionally, the emphasis has been on components as binary entities of independent third-party composition. The distributed computing environment of the Internet, however, has introduced a new and important dimension of *dynamism*: not only are prefabricated components used as building blocks of larger systems, but they arrive at their destination on runtime, by programmatically shipping them across the network. Applets and downloaded plug-ins are typical examples. In addition, components often need to interact across the network via a persistent channel of communication. Messenger systems are a good example. Most importantly, these dynamic aspects of code-shipping and cross-network interaction introduce *security* as a major concern in component technology.

In existing technologies—including COM, Corba, and Java Beans—components interact via the *client-server* model: one component (the client) obtains on runtime a handle

to another (the server), and then uses it to invoke operations via interfaces, see Fig. 1(a). This kind of interaction is asymmetric (client versus server) and explicit (client calls operation on server). The client-server model has been successfully applied to numerous communication-related disciplines of computer science.

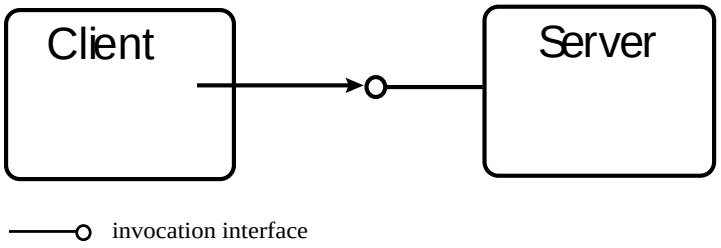
There is, however, a different model of interaction, which we will call the *persistent link* model, see Fig 1(b). In this model, components must be linked to one another before they can interact. Each component is developed assuming some *imports* given. These imports are used as if they were actually defined inside the component. At link time, the imports are linked to the other component's *exports*. From this time on, calls to an import refer to the linked export in the other component. This kind of interaction is symmetric and implicit: references to imports do not look any different than references to local entities. The persistent link model is usually used for modules, or packages, which are not runtime abstractions.

In this paper we propose a new component model that addresses the dynamic nature of Internet components by supporting the client-server and the persistent link modes of interaction in one streamlined architecture, see Fig. 2. It is based around the concept of a *cell* (the name is inspired by the biological notion of cell). Cells are first-class stateful entities which exist in the runtime environment, and which offer two kinds of services: (1) operation invocation via *services* (like COM/CORBA/Java objects), and (2) persistent link to other cells via *connectors* (like modules). Linking is totally dynamic and revocable: two linked cells can be unlinked at a later time. Cells are meant as a new programming abstraction – a language mechanism with different responsibilities than classes and objects.

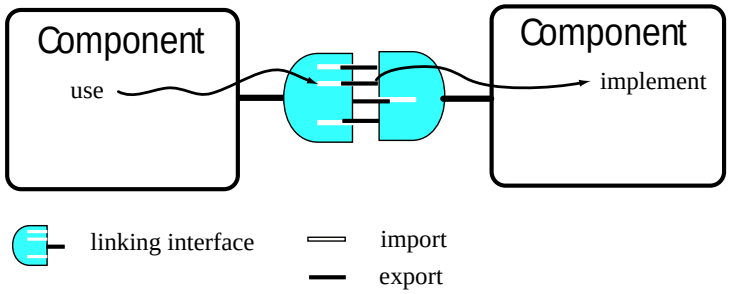
As illustrated in Fig 2, a cell contains both passive code, in the form of classes and functions, and executing stateful elements in the form of objects. This is another focus of the model: cells are runtime *object containers*, and mechanisms are provided to control the level of isolation of a cell with respect to its objects. Some important features of cells are outlined below.

Per-link security mechanism. Cells authenticate each-other at link time, and the linking succeeds only if it is agreed by both sides. Interaction then usually proceeds without further checks. For heavy interactions over a relatively short period of time, this kind of protocol is more efficient than per-invocation security checks. Such interactions are typical for Internet components: applets usually arrive for a short period of time, and components connecting across the network do so for a limited time. In both cases, appropriate security checks are vital. Since per-link checks occur only once at connection time, more substantial, possibly time consuming, policies can be devised. Per-call checks on the other hand must typically be light.

Runtime object containers. Every object lives in some cell, and so cells divide the general object space into smaller groups of related objects. Security-wise, a cell can protect its objects in varying degrees: several parameter passing mechanisms are provided to help control the level of isolation of a cell with respect to its objects. In addition, a cell-serialization operation is supported, which allows the set of objects belonging



(a) The client-server interaction model



(b) The persistent link interaction model

Figure 1: Interaction models

to one cell to be serialized as a group.

Cross-component inheritance. A class in one cell can extend a class in another, simply by extending an import. This cannot be supported in pure client-server component models because inheritance is not invocation. Since cells can be unlinked and then relinked, the superclass of a class extending an import may dynamically change on runtime. The syntactic fragile base-class problem does not apply because imports within connectors are *statically typed*, so the superclass type cannot change. Cross-component inheritance is central to Internet programming: any applet-like shipped component potentially needs to inherit from a local system component (a library).

Natural system architecture. In many cases, symmetric composition reflects the natural structure of the system at hand, resulting in a better architecture. For example, a messenger system is naturally built as two components interacting symmetrically across the network. In fact, the linking operation gives a concrete meaning to component *composition*. In COM/Corba/Java, which support only the client-server model, composition is more like a metaphor which in practice translates to the two components being clients of one another over a period of time.

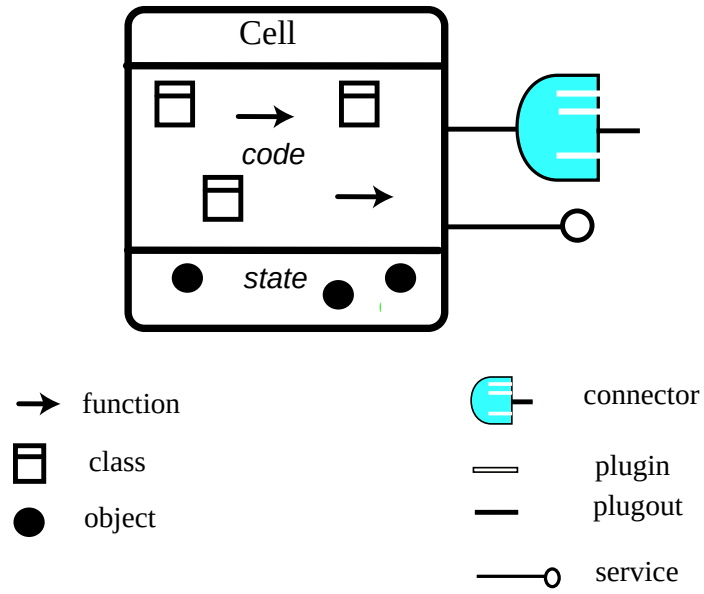


Figure 2: A Cell

Cross-network link. On top of the basic concepts, it is possible to built a distribution layer for cells. The move from the local cell model to the distributed model is analogous to the move from COM to DCOM; but whereas distribution in DCOM means remote invocations, here it also means cross-network cell linking (in addition to remote invocations). Cells can directly link across the network, just by applying the cell linking operation between cells on different nodes. Such persistent inter-node connections bear a strong analogy to the persistent connections of internet protocols such as ftp and ssh, but are at a higher level of programming abstraction. What this means is that persistent service connections, which must be performed in an ad-hoc manner in other languages, can be coded directly in the cell language.

Unification of concepts. In existing technologies, the term "component" actually refers to two distinct entities: the runtime invocable instances (COM/Corba/Java objects) and the classes from which these instances are created (COM/Corba/Java classes). In order for one COM object to call an operation on another COM object, their DLLs containing classes in binary form must first be linked by the operating system. It is a general-purpose kind of link that lies outside the scope of the component model. Cells in contrast are at the same time invocable and linkable entities, and the linking mechanism is an integral part of the model. In addition

to simplicity and minimality of concepts, this has an important advantage in terms of security: in the authentication protocol for linking cells, each cell may query the other via its services. This way, the authentication protocol may rely on substantial knowledge directly obtained from the potential security threat. In Java, security checks are per class, not per instance, so only general external information such as where the class came from or who signed it can be considered. The Java security protocols cannot *query* the potential security threat.

The cell project focuses on both foundations and implementation. We have recently launched a programming effort aiming to implement cells as an extension of Java. In this short paper we sketch the constructs and protocols supported by cells, via a series of examples.

2 The Cell Model

We first illustrate the basic concepts, and then discuss some more advanced topics, including distribution. The examples in this section are written in a Java-like language with cells.

2.1 Basic Cell Concepts

Cells are containers of classes, functions, and objects, which interact via explicitly declared external hooks (Fig. 2). There are two kinds of hooks – *connectors* and *services* – through which two modes of interaction are supported:

1. Seamless interaction following linking via a connector, a module-style connection; and,
2. Explicit interaction via a particular service, a client-server, COM/Corba/Java-style interaction.

Two cells may interact seamlessly (to be explained in a moment) after they are linked via a connector. A connector on a cell lists a series of imports and exports, the *plugins* and *plugouts*. Each plugout is implemented by an appropriate element contained in the cell: a *class plugout* is implemented by a contained class, and a *function plugout* is implemented by a contained function.

To interact with a cell via connector I, another cell must be *linked at* I. That cell must also have a dual connector named I, in which the plugins and plugouts trade places. Once a linkage between two cells is established, references to a plugin in a cell refer to the connected plugout in the other cell. A call to a function plugin within a cell looks like a call to an external function whose implementation will be provided via linking. Similarly, extending or newing a class plugin looks like extending or newing an external class to be provided via linking. Interaction via connectors is therefore seamless from the point of view of the cell authors. This notion of linking via connector matching is close to module interface matching, but connectors are more dynamic: two linked cells may be unlinked at a later time.

Class plugins allow a class in one cell to inherit a class in another cell. This powerful feature allows, for instance, shipped code to inherit from local libraries, as is the case with applets using local libraries via inheritance (we illustrate how to implement applets with cells later).

The second mode of interaction is via services. A service lists a series of *operations* which can be invoked by clients. In contrast with plugins and plugouts, ser-

vices may only contain operations: no classes are allowed. Every operation is implemented by a function contained in the cell. Clients invoke a operation on a service by first obtaining an handle to the service, and then invoking the operation on the handle, much like in COM (where *interfaces* are the parallel for our services). A given cell can have both connectors and services, some of each.

Security-wise, cells are like walls protecting their contained elements. The internals of a cell cannot be accessed directly, only through connectors and services. As explained in the introduction, cells support a per-link security mechanism: at link-time, the two cells may authenticate each-other, and decide whether or not to allow the link. This is realized by a special per-connector *trust* predicate that allows arbitrary security checks to be coded. This is illustrated below.

Example: Explicit interaction via services. A first example shows how a COM-like purely service-oriented cell can be defined. The cell has no connectors.

```
cell nameserver {
  service NameServer {
    int lookup (String);
    void setRoot (String);
  }

// internal elements

    Root theRoot = new Root();  // cell field

    class Root {   // class Root
        String root;
        ...
    }

    // internal function

    String toLowerCase(String s) {...}

    // implementation of operation lookup:
    // look up a name, returning its number

    int lookup (String s) {...}

    // implementation of setRoot:
    // set default domain

    void setRoot (String newRoot) {
        theRoot.root = newRoot;
    }

}
```

In this example, the nameserver cell has one service – `NameServer` – with two services: `lookup` and `setRoot`. Their implementations is given inside the cell. The cell also contains a private function, `toLowerCase`, which these implementations can invoke, and a field `theRoot`, which is an object representing the current root value. The presence of objects in cells is important because it allows cells to have state. Within the cell, the private elements can be referred to directly by name.

Continuing with this example, the following client invokes some operations on that service.

```
// obtain reference to the cell via some
// naming service

nameserver = lookup("nameserver");

// invoke operations

(nameserver  ◁ NameServer).setRoot(".jhu.edu");
(nameserver  ◁ NameServer).lookup("cs");
```

Example: secured linking via a connector. The next example repeats the same functionality as the previous one, but using a connector instead of a service to explicitly contrast the differences between the two. In addition, the server contains a *trust* predicate on the `NameServer` connector to restrict the parties that can link via that connector.

```
cell linkingnamesvr {
  connector NameServer {
    plugouts {
       int lookup (String);
       void setRoot (String);
    }

    // trust predicate: x represents the
    // other cell connecting to this one
```

```
    trust(x) {
        //get certificate from other cell
        String cert = (x ◁ Cert).certificate();
        ... check that cert is a valid
        certificate for access. Return true
        iff the certificate is valid ...
    }
  } // end connector NameServer

  // implementation of lookup and setRoot as in
  // nameserver above ...

} // end cell linkingnamesvr

cell nameuser {
  connector NameServer
    plugins {
        int lookup (String);
        void setRoot (String);
    }

    service Cert {
        String certificate();
    }
  }


  // implementation of certificate: return the
  // certificate for this cell ...

  String certificate ...

  // somewhere inside the cell, we may have
  // seamless calls to setRoot and lookup:

    ... setRoot("jhu.edu") ...
    ... lookup("cs") ...
    ... lookup("www") ...

} end cell nameuser

// obtain reference to nameuser
// via some naming service

nameuser = lookup("nameuser");

//obtain reference to linkingnamesvr

linkingnamesvr = lookup("linkingnamesvr");

// link cells

link nameuser linkingnamesvr at NameServer;

// unlink at a later time

unlink nameuser linkingnamesvr at NameServer;
```

In this example, the `linkingnamesvr` cell is similar to the `nameserver` above, but we use a connector `NameServer` with plugouts instead of a service with operations. The user cell `nameserver` also has a connector `NameServer`, but with a dual set of plugins.

The two are dynamically linked by the `link` statement. The `setRoot` and `lookup` operations can be directly invoked from inside `nameuser`, because those services are plugged in. If `lookup` were invoked before the connection to the name server was established, a runtime error would result.

The `linkingnamesvr` cell above includes a trust predicate on the `NameServer` connector. The `nameuser` cell includes a service interface `Cert` which the `linkingnamesvr` uses in its trust predicate to determine if the user is authorized to link via `NameServer`. The `linkingnamesvr-nameuser` linking succeeds only if the trust predicate returns `true`. (In the general case, `nameuser` could also have its own trust predicate on the `NameServer` connector for mutual authentication.)

2.2 Cells as Object Containers

Cells *own* the objects they create. With this axiom we can define adequate notions of inter-cell parameter passing, inter-cell object reference, and serialization. These notions provide a design space in which the degree of isolation of a given cell may be controlled. They also lead to an elegant generalization to a distributed architecture, which requires few additional concepts.

Parameter Passing. One of the more difficult design decisions is how cells can talk to one another at the lower (object) level. If no thought were given, one cell could own an object that directly points to an object owned by another cell, and such a channel may persist even after the connection between the two

cells has ended. This constitutes a potentially risky back-door. We thus define three parameter passing mechanisms for how object arguments are passed between cells:

1. by (hard) reference. Using this mechanism, a cell can get hold of a reference pointing directly to an object owned by a different cell. Such pointers are efficient but violate cell-encapsulation. They should be used only between tightly coupled cells, where a lower degree of isolation suffices.

2. by `copy`: the object and all objects it refers to, transitively, are copied. When an object is copied from one cell to another, its class must be either defined in, or plugged into, the target cell. This mechanism guarantees full isolation, but may involve copying large segments of data.

3. by `modulated` reference: this is an object reference which is modulated by the cell, i.e. the reference is sustained by the approval of the owning cell, and approval can in the future be withdrawn. See Fig. 3. Modulated references can span virtual machines and locations.

Modulated references are a variation on a well-known concept in distributed object systems: rather than holding a hard reference to an object, the cell holds a stub object which knows how to reach the actual object which is in another cell. This idea normally applies to objects at different physical locations, but we generalize it to objects in different cells.

Modulated references allow one cell to hold a logical reference to an object in another cell without violating its encapsulation. The accessability of such pointers is subject to the connection: a modulated reference passed through a connection is only "live" as long as the connection itself is live. Objects held as direct pointers on the other hand will stay alive even if their defining connector has been disconnected. Modulated references generalize naturally to cross-network references, where clearly direct pointers are impossible.

Cell Serialization. Another important aspect of cells we are building in is a serialization protocol. Object serialization is e.g. in Java an extremely *ad-hoc* operation—much manual notation is needed to specify how deep to serialize the object structure, and there is a huge code versioning problem to deal with when objects are deserialized. We believe serialization however *can* effectively happen at a conceptually higher level than individual objects, namely at the cell level. Each cell is a natural encapsulation of its code and its owned objects, which can be serialized as a unit, and later deserialized. The serialization protocol is straightforward except for hard references to nonlocal objects: since the object pointed to is not owned by the cell, it cannot be serialized. One of three choices is possible: raise an exception, replace the object with null, and turn the reference into a modulated reference.

2.3 Cells in Distributed Environments

As illustrated in the previous section, cells can be used in a local setting for modular and secure programming. However, a main focus is for cells to serve as a cross-network abstraction for distributed environments. This means that cells are global entities: every cell has

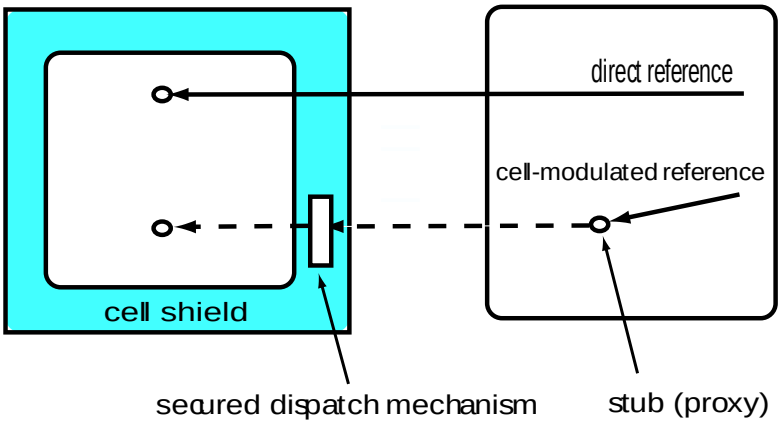


Figure 3: direct versus cell-modulated references

an identity, its *cell identifier (cid)*, which is unique over the network (like COM GUIDs). References to cells are valid across the network, and reference holders use them in the same way regardless of whether they refer to local or to remote cells. This is the basis for the following supported functionality:

1. Cells may be copied and shipped across the network, to be linked/invoked locally (applet-style scenario).

2. Cells may move from one network location to another (a form of roving agents).

3. Cells may be linked across the network, that is a cell in one location can be linked to a cell in another location via a plugging interface.

4. A client may transparently invoke operations on a remote cell's service.

Copying a cell from one location to another realizes the applet-style mechanism in a way that allows for systematic authorization and invocation on the client side. Linking cells across the network allows for persistent peer-to-peer connections to be made at a high level of abstraction rather than at the low level of internet protocols. Consequently, instead of performing security checks on a per-operation-invocation basis, such checks will typically be performed on a one-time per-link basis. This is a crucial point for security-sensitive distributed

applications that need to maintain reasonable performance.

We now elaborate on some of the details involved in the design. The network is set of *locations*, each with an associated address space. Every cell must exist in some location. Cells can communicate across the network via connectors or services in a way quite similar to local interaction. This is one of the key points of our design: networked component interaction is a variation of local component interaction. In particular, the mechanisms for passing object parameters between cells on different locations are a subset of those for passing objects between cells in one location: objects can be passed by copy or by modulated reference. As for local cells, a copied object must have its class present in the target cell, either defined or plugged-in. Obviously, objects cannot be passed across the network as direct references.

Copying (moving) a cell means serializing its static code part, i.e., classes and functions declared inside, as well as its current state, i.e., the objects to which it holds a reference, each in its current state. Object serialization is governed by ownership as described in 2.2.

Cell-cell Connections which span the network must be a restricted form of connection. Objects must always run locally, and the class for the object and all its superclasses must be in the same location as the object itself. For this reason, one cell cannot plug in classes from another cell across the network. Furthermore, as mentioned above, parameters passed through a networked cell-cell connection must be by copy or by modulated reference. Connectors with such a restriction are declared `networked`.

Example: Cross-Network Linking. Consider the `linkingnamesvr` example above. This example can be explicitly implemented with the user and server cells being on different locations with minimal change in the code. First, the connector `NameServer` in both cells must defined as `networked`. Then, supposing a reference for the server cell has been obtained in the user's location, the exact same `link` command is coded, and the interaction proceeds transparently over the network. The service example `nameserver` can also be made distributed, just by defining the service `NameServer` as `networked`.

Example: Applets. Applets are cells that come from other locations on the network. In Java, an applet arrives at a location with requirements for several system libraries, such as `java.awt` and `java.applet`. In a cell implementation of the applet concept, there must be a cell for each such system library, and the applet will then negotiate a linking with such a library. Applets with insufficient authorization will not be allowed to plug into the system libraries. This structure is illustrated in Fig. 4. The library cells in the client's location are of the form

```
cell awtlib {
  connector AWTRoot {
    plugouts {
        class Canvas {...}
        ... other java.awt root class headers ...
    }
    // authorization policy to be enforced on
    // library users connecting via AWTRoot
    trust(x) {...}
  }
  connector AWTEvent {
    plugouts {
    ... the java.awt.event class headers ...
    }
    // authorization policy to be enforced on
    // library users connecting via AWTREvent
    trust(x) {...}

  }

  ... definition of all AWT classes ...
```

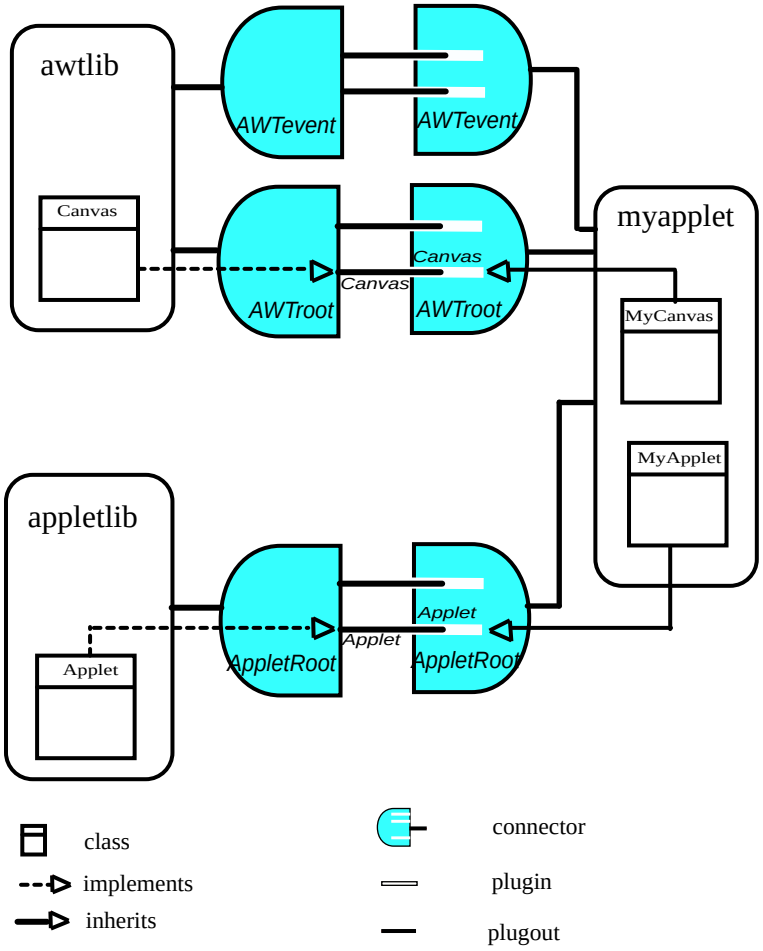


Figure 4: Applets with cells

```
} // end cell awtlib

cell appletlib {
  connector AppletRoot {
    plugouts {
        Class Applet {...}
        ... other java.applet class headers ...
    }
    // authorization policy for applet library
    // users connecting via AppletRoot
    trust(x)  {...}

  }

  ... definition for all applet classes ...

} // end cell appletlib
```

In the server's location, an applet cell of the following form is declared:

```
cell myapplet {
  // import all the relevant libraries
  connector AWTRoot {
    plugins {
        class Canvas {...}
        ...
    }
  }

  connector AWTEvent {
     plugins {
     ... the java.awt.event class headers ...
     }
  }
```

```
  connector AppletRoot{
    plugins .{
        class Applet {...}
        ...
    }
  }

  service Run {
    void main() {...}
  }

// internal classes

  // Applet imported from AppletRoot
  class MyApplet extends Applet {...}

  // Canvas imported from AWTRoot
  class MyCanvas extends Canvas {...}

  // implementation of main
  void main() {...}
} // end cell myapplet
```

Assuming the client obtained a reference for that cell, stored in `remoteapplet`, it could copy it and then use it locally. After connecting to the client's local library cells, the plugins `Applet` and `Canvas` will refer to these local classes. The following code copies and then links the applet dynamically with the libraries:

```
// link
myapplet = copy(remoteapplet);
link myapplet awtlib at AWTRoot;
link myapplet awtlib at AWTEvent;
link myapplet appletlib at AppletRoot;
// run the applet
(myapplet ◁ Run).main();
```

Note that the library cells `awtlib` and `appletlib` may themselves hold state information. For instance, the `appletlib` cell could keep track of the applets currently linked to it, and keep an audit trail of critical library functions accessed. Also note that there is no need to declare any `networked` interfaces here since all connections are made locally (after copying).

One important feature of this example is that it shows how a class in one cell can inherit from a class in another cell. When a `MyApplet` object is created, some of its code is in the `myapplet` cell, and some is in the `appletlib` cell due to inheritance from `Applet`. This does not violate the requirement that objects have their code locally, since all cells involved are at the same location.

In addition to code, the imported applet could contain some data, as part of its state. For instance, it could contain relevant web links dynamically kept up-to-date in the server's location. The server could even pick links which are appropriate to the target client.

3 Related Work

We have already given the broad picture of how cells relate to existing component technologies in the previous sections; here we briefly discuss related work in the component research community.

The importance of modeling the dynamic aspect of component connections has been addressed by several researchers. Connection-oriented programming as discussed in [Szy98] is closely related to our plugging interface connections, which can be viewed as a concrete realization of that idea.

There are several foundational component frameworks that have been developed. Piccola [AN00, LAN00] is based on a core calculus for components. It is a glue scripting langauge for putting together software. Importantly, it explicitly models the dynamic nature of component connection. It does not include 1-1 linking/unlinking via plugging/unplugging or trust relationships and is untyped. Several purely static component calculi have also been developed. [SC00] models typed static component composition. Some module systems are also very close to static component systems; one such system is Units [FF98], which allows some dynamic module linking but is lacking any service in-

terfaces or statefulness.

Further information on the cell project may be found at www.cs.jhu.edu/hog/cells.

References

[AN00] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2000. to appear.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[LAN00] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[SC00] Joao Costa Seco and Luis Caires. A basic model of typed components. In *ECOOP*, pages 108–128, 2000.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.