Towards Self-Organizing Distributed Computing Frameworks: The H2O Approach

Dawid Kurzyniec, Tomasz Wrzosek, Dominik Drzewiecki and Vaidy Sunderam
Dept. of Math and Computer Science, Emory University, Atlanta, GA 30322, USA
{dawidk,yrd,drzewo,vss}@mathcs.emory.edu

Abstract

A novel component-based, service-oriented framework for distributed metacomputing is described. Adopting a provider-centric view of resource sharing, this framework emphasizes lightweight software infrastructures that maintain minimal state, and interface to current and emerging distributed computing standards. In this model, resource owners host a software backplane onto which owners, clients, or third-party resellers may load components or component-suites that deliver value added services without compromising owner security or control. Standards-based descriptions of services facilitate publication and discovery via established schemes. The architecture of the container framework, design of components, security and access control schemes, and preliminary experiences are described in this paper.

1 Introduction

The benefits of distributed resource sharing are well established, and numerous software architectures and toolkits have evolved in recent years to support this mode of computing. Some, such as UD cancer research [27] and SETI@Home [21] cater to specialized (classes of) applications, while others, including MPICH [10] and PVM [8] are generally confined to single administrative domains. More generalized metacomputing systems, or *grids*, have gained tremendous popularity in recent times as enabling secure, coordinated, resource sharing across multiple administrative domains, networks, and institutions within the context of a central abstraction called a *virtual organization*. This metacomputing model [6] has been realized in several software toolkits including Globus [5], Legion [19], and Globe [28], and has been deployed very successfully in a number of application domains [9, 18].

The virtual organization model makes collaboration between large and geographically distributed organizations much easier and more effective. It has proven to be very well suited for grand challenge scientific problems [1]. However, we believe that a more lightweight and stateless model may be more suitable for applications of smaller magnitude, involving individuals and small organizations that may want to share their resources on a peer-to-peer basis rather than with explicit coordination and centralized services for authentication, registration, and resource brokering. In this paper, we propose such a complementary architecture and software platform to enable lightweight, reconfigurable, provider-centric distributed resource sharing. This framework, tentatively labelled H2O, is based on the premise that resource providers act locally and independently of each other and of clients, and that by minimizing or even eliminating coordination middleware and global state at the low level, self-organizing distributed computing systems can be enabled. In the H2O model, a software backplane within each resource supports component-based services that are completely under owner control; yet, authorized clients may configure and securely use each resource through components that deliver compute, data, and application services. Such an architecture results in significantly reduced state at lower levels of the system, thereby resulting in increased failure resilience and dynamic adaptability. The H2O architectural model, proposed

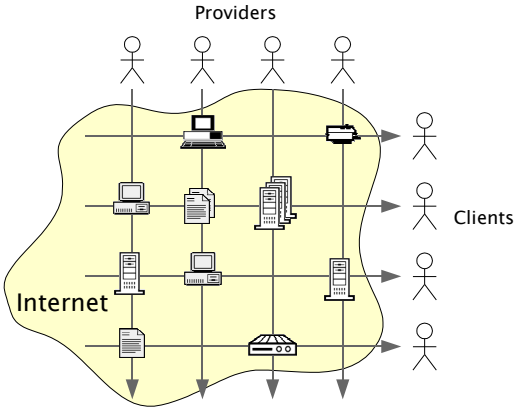


Figure 1: Resources, providers, and clients

approaches to dealing with the issues of security, resource management, service interfaces and component interaction, salient features of a prototype implementation, and preliminary experiences are presented in this paper.

2 H2O Architecture

The overarching goal in the design of H2O is to minimize distributed state in the middleware that facilitates distributed resource sharing. Particularly in frameworks that span multiple administrative domains (including the extreme case of peer-to-peer systems), autonomy and loose coupling are crucial to delivering flexible, dynamic, distributed computing platforms. Our proposed approach to realizing this model is outlined in the following subsections.

2.1 Components, Containers, and DVMs

The high-level model we assume consists of a dynamic collection of *providers* who *independently* make resources available over the Internet, and a dynamic set of clients who discover, locate, and utilize these resources. This model is illustrated schematically in Figure 1, which highlights the independence of the entities involved and the absence of explicit coordination middleware that binds resources, providers, and users. Thus, at the lowest level, resource sharing occurs between a single provider and a single client, enabling every such pairing to tailor their interactions exactly as appropriate, e.g. choosing identity formats, resource allocation, and compensation agreements. Further, since clients may themselves be providers (and vice versa), cascading pairwise relationships may be formed. The abstraction of a *distributed computing platform* is pushed as close to the client as possible; this strategy reduces state and the degree of coupling without compromising functionality.

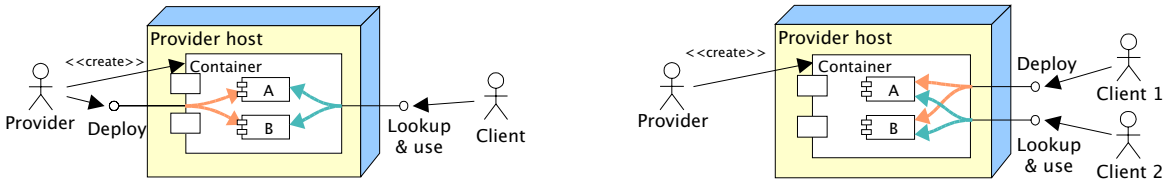
Our framework assumes that each individual resource may be represented by a software component that provides services through well defined remote interfaces. The model leverages (but does not mandate) Web Service technologies [30] through the use of WSDL [3] as a component description language. In H2O, providers supply a runtime environment in the form of a *component container*. Current systems based on a service hosting model [24, 15] assume that services are static, and that

they are deployed exclusively by the owner of a container. Such an approach, though appropriate in electronic commerce and similar domains, does not support resource *sharing*. In contrast, the containers proposed in our framework will be capable of hosting dynamic components that are supplied and deployed by (authorized) external entities, thereby facilitating (re)configuration of services according to client needs; this distinction is highlighted in Figure 2. A simple example scenario might involve a data center providing hardware resources, and a third-party enterprise supplying optimized linear algebra packages, to a simulation end-user.

Metacomputing applications typically benefit from an aggregated *context* in which concurrent computing activities take place, usually in the form of an explicit or implicit *distributed virtual machine* (DVM). To provide such context, our framework introduces the notion of a *distributed component container*, meaningful only as a client side abstraction. It is realized through a special collection of components that execute on individual resources and coordinate to provide a consistent projection of a virtual machine. That network of *DVM-enabling* components alone maintain hard distributed state; by isolating state to a single component-type, flexibility and independence can be largely preserved while constructing a consistent distributed computing platform for client applications. This architectural model is depicted in Figure 3.

2.2 Operational Overview

In a canonical H2O operational scenario, resource owners execute component containers on their resources, and specify fine grained constraints on: the types of service components that the container may host, the identities or categories of clients that may avail of these resources, level of access, priorities, and other attributes. Components that present a service interface, either independently or by utilizing other services, may be loaded into containers by providers, authorized clients, or third-party resellers, as shown in Figure 4. Client applications then access these services; runtime support provided by the container handles service delivery, lifetime management, and constraint enforcement. Matching providers with clients is accomplished via orthogonal schemes, including basic registration and discovery mechanisms provided by the H2O system itself; the most appropriate solution may be chosen for a given situation. For example, small collaborative groups may use well-known container end-points, departmental clusters may use designated, centralized registration and lookup services, and world-wide UDDI registries may be used for global resource sharing.



(a) traditional model

(b) proposed model

Figure 2: Conceptual abstraction of a container

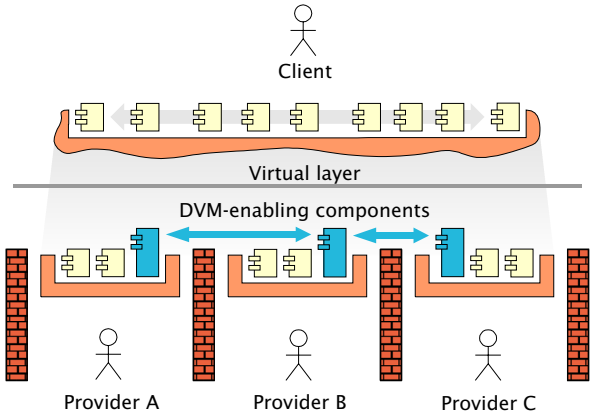


Figure 3: Distributed component container

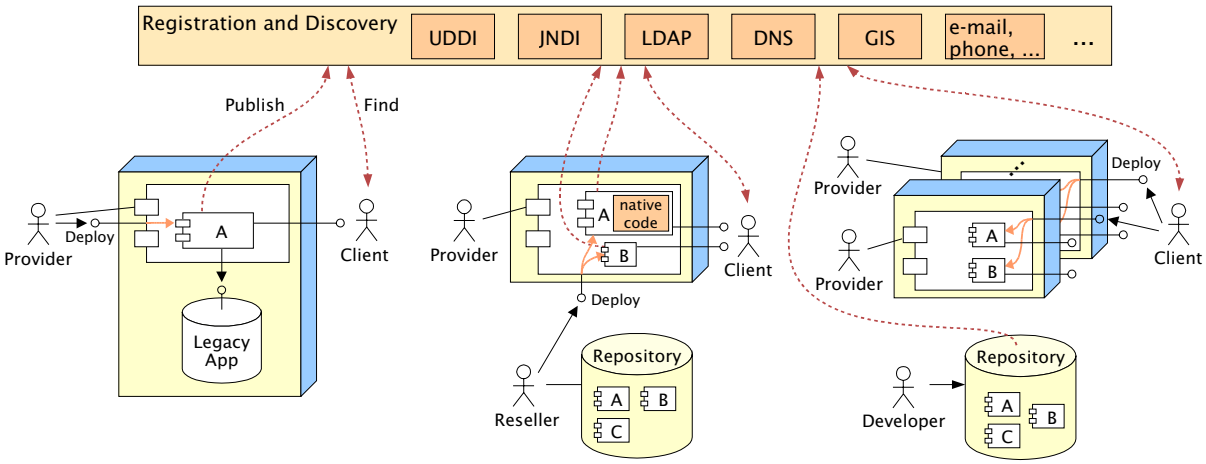
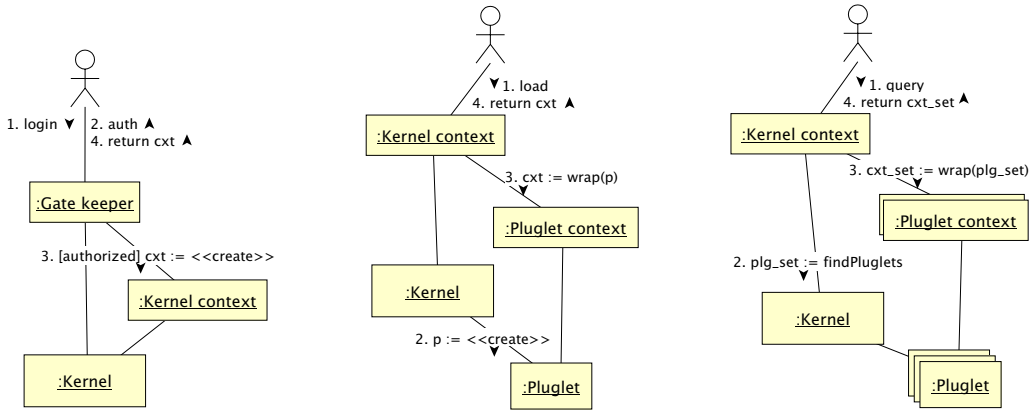


Figure 4: Various example usage scenarios

3 Prototype Design and Implementation

The H2O framework is currently in the phase of alpha testing. The system consists of the prototype container implementation and a repertoire of general purpose and exemplary components. We envision that users, installations, and third-party providers would build libraries of components in the long term as needed. The container implementation leverages Java technologies, resulting in immediate benefits of platform independence, a built-in security infrastructure, and a uniform component format. However, the Java-based H2O container is capable of hosting not only Java-based but also native services adapted to H2O using JNI [13] interfaces. Furthermore, these services are accessible by heterogeneous clients (implemented in any language on any platform) that may communicate with the service providers using a variety of wire protocols.



(a) login (b) load pluglets (c) find and access pluglets

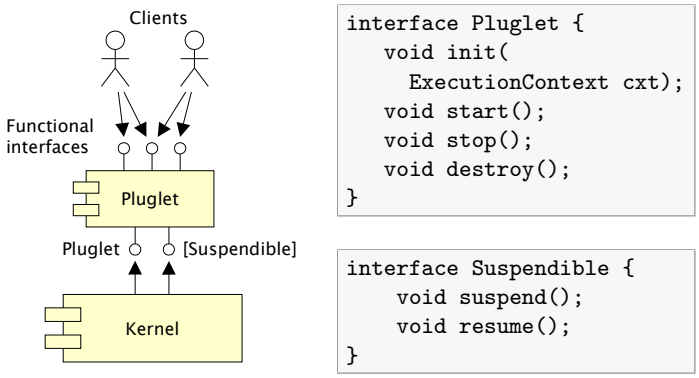
Figure 5: Interaction between client and kernel

3.1 Kernel and Pluglet Abstraction

To distinguish from other component frameworks, we use the terms *kernel* to denote an H2O component container and *pluglet* to refer to an H2O component. The term *pluglet* reflects similarities between H2O components and Java *applets* and *servlets* that execute in client and server contexts respectively; H2O pluglets are components hosted by a provider kernel but may be deployed, or *plugged in*, by the client, the provider, or third-party resellers.

The first step in installing or accessing components within a container involves authentication, as shown in Figure 5(a). A *gate keeper* verifies the client's security credentials and returns an instance of a *kernel context* through which further actions may be taken. Authorized clients may load new pluglets into a kernel (Fig. 5(b)), specifying the location (URL path) of pluglet executable files, the pluglet class name, initialization parameters, and optional meta-information that may later be used for lookup and discovery purposes by other clients. Clients may also lookup and access pluglets that have been already loaded (Fig. 5(c)), by furnishing an appropriate filtering query. As a result of loading or locating pluglets, clients receive *pluglet contexts* that may be used to access pluglet meta-information or perform meta-operations (e.g., destroy pluglets), in addition to invoking operations defined by the pluglet itself.

As shown in Figure 6, pluglets mandatorily implement the `Pluglet` interface, and may optionally implement the `Suspendible` interface. These interfaces are intended for use exclusively by the kernel to notify the pluglet of imminent state changes. Clients access services provided by the pluglet through *functional interfaces* that may and generally are provided in addition to these two. Methods of the `Pluglet` and `Suspendible` interfaces are invoked in a guaranteed order, according to the following regular expression: "*init (start (suspend resume)* suspend? stop)? destroy*". Upon initialization, the pluglet is presented with its *execution context* containing introspective information – e.g., regarding the identity of the client that loaded it, along with any pertinent metadata. The *start* method notifies the pluglet that it is about to be activated; this enables the pluglet to receive remote calls and to create new threads of execution. An active pluglet may be suspended,




```
interface Pluglet {
   void init(
     ExecutionContext cxt);
   void start();
   void stop();
   void destroy();
}
```

```
interface Suspendible {
    void suspend();
    void resume();
}
```

Figure 6: Pluglet interfaces

resumed or terminated by authorized users or by the kernel itself, the latter as a result of the resource owner's access control policy. Pluglets implementing the `Suspendible` interface are notified before suspension and after resumption with the `suspend()` and `resume()` methods, respectively; they may take this opportunity to migrate their workload or perform other actions. Similarly, pluglets are notified before termination via the *stop* and *destroy* methods. A possible usage scenario is to propagate this notification to other interested parties and to perform a clean shutdown. Invocations of *suspend*, *stop* and *destroy* methods are guarded by timeouts, so the pluglet should execute these methods as quickly as possible to avoid forcible suspension/termination.

3.2 Component Interaction

One of the key features of the H2O container is that it provides hosted services (i.e. pluglets) with facilities for effective and structured communication. This feature is specifically targeted towards distributed applications, making their development easier and more efficient. The simplicity of this model is illustrated in Figures 7 and 8 that show the interface definition and the implementation of a canonical H2O pluglet, respectively. We have chosen remote method invocations as the native model of inter-component communication due to the object-oriented nature of the H2O framework. However, in the interest of flexibility, we elected to introduce several extensions to the original Java RMI model. In addition to synchronous calls, H2O supports asynchronous calls (results being returned as *futures* [4]), and one-way calls (that may effectively be used for message passing). Also, the underlying wire protocol, the data serialization semantics and the quality of service are not fixed (as in traditional RMI) but they may be chosen dynamically according to the given situation. This allows for better coupling of various components under different circumstances; e.g. scientific simulation components may communicate via specialized Myrinet-based protocols but they may, at the same time, expose SOAP-based control interfaces for lightweight clients. The appropriate protocol binding is chosen by the client (which may be another pluglet) at the run time, as shown in the Figure 9.

H2O inherits these communication capabilities from the underlying RMIX communication substrate. RMIX [14] is an extension and generalization of Java RMI, and it provides a common framework and semantics over a variety of method invocation protocols. Specific protocols are independently supported by pluggable service provider modules that enable dynamic (run-time) extensibility of the communication layer. Additionally, RMIX offers several enhancements to the

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String hello(String name)
           throws RemoteException;
}
```

Figure 7: "Hello" pluglet interface

```
public class HelloImpl implements Hello, Pluglet
{
    public void init(ExecutionContext cxt) {}
    public void start() {}
    public void stop() {}
    public void destroy() {}

    public String hello(String name) {
        return "Hello " + name + "!";
    }
}
```

Figure 8: "Hello" pluglet implementation

```
String PROTOCOL = "edu.emory.mathcs.rmix." +
                  "spi.jrmpx.JrmpxProvider";
Object proxy = plugletCxt.getPluglet(PROTOCOL);
// the proxy implements the same set of
// remote interfaces as the remote object
Hello hello = (Hello)proxy;
// now, simply invoke the remote method
String greeting = hello.hello("Client");
```

Figure 9: Use of "Hello" pluglet

RMI communication model and implementation, like the aforementioned asynchronous and one-way calls, dynamic stubs that enable run-time binding, and customizable virtual endpoints that allow the same remote object to be accessed via different protocols and/or with different access restrictions. Currently, there are two protocol provider modules available: RMIX-JRMPX, based on standard Java RMI and Java serialization, and RMIX-XSOAP, based on XSOAP [22] and using SOAP as a wire protocol. RMIX-JRMPX allows pluglets to achieve maximum compatibility and exploit the full semantics of Java RMI, like distributed garbage collection and remote class loading. On the other hand, RMIX-XSOAP provides maximum interoperability, enabling H2O containers to expose hosted pluglets as Web Services. To make the picture complete, we are currently working on a new provider module that will bypass some serialization features and sacrifice interoperability to gain the highest possible communication performance in environments such as homogeneous tightly-coupled clusters.

Although RMIX is the default communication substrate for H2O components, there is no limitation nor compulsion, as pluglets may choose alternative means of communication, including direct

socket connections or third-party communication libraries, subject to the resource owner's access control policy.

3.3 Security

In keeping with the principle of provider-centricity, the default H2O security mechanisms are located within component containers, i.e. the H2O kernel. The gate keeper (Fig. 5(a)) provides single sign-on facilities at the kernel level in a manner that is conceptually similar to the `rsh` daemon. Clients always access H2O kernels and pluglets through opaque security proxies that contain that client's identity information as established by the gate keeper. Security proxies are able to perform authorization decisions based on policies specified by the kernel owner and the pluglet *loading entity* (the user who loaded the pluglet). All pluglet code is executed within a thread group controlled by the kernel. Code invoked explicitly or implicitly from the kernel (through the `Pluglet` or `Suspendible` interfaces) executes within the loading entity's security context. On the other hand, remote calls are executed within the context of a *discovering entity* (the user who *obtained* that particular reference). A pluglet may modify these default semantics to a certain extent (obviously, modifications may never result in broader access than that possessed by the loading and discovering entities together). The discovering entity may choose to share a pluglet reference in one of two ways: by subletting its security context, or by requiring the delegate to establish its own context by logging in to the kernel itself.

Authentication in the H2O prototype is defined in terms of the Java Authentication and Authorization Service (JAAS) [25] that supports an extensible authentication model based on Pluggable Authentication Modules (PAM) [26]. Every client may have multiple identities, any subset of which may be used by any kernel. The use of X.509 certificates is recommended, but kernel providers may configure authentication modules according to their own preferences, depending on the set of users (clients) being targeted. To make pluglets capable of acting on behalf of users, e.g. to load other pluglets, the H2O project will implement the outcomes of the IETF standardization effort for restricted X.509 proxies [11].

There are two issues that need to be addressed while considering the authentication process: passing credentials securely to the kernel and verifying the provided identity credentials against some repository. While the latter is handled through JAAS Pluggable Authentication Modules, the former aspect must be designed so that a kernel client is able to provide his or her credentials in a protocol-independent manner. In addition, a kernel should be able to chose an appropriate authentication method, recognize the client authentication message and decode it, regardless of what particular type of credentials were passed. There is a standard emerging [16] that we believe might be employed for secure and reliable credential exchange over SOAP, thus facilitating access to the H2O platform from different types of clients. We are currently investigating its applicability.

Authorization in H2O has two aspects: (1) protecting and managing resources owned by a kernel provider, and (2) controlling access to pluglet instances according to the policies of loading entities. The first issue is directly addressed by the Java security model and JAAS that together allow for fine grained control of system resource usage based on a client's identity. The second issue is addressed at the level of remote interfaces, i.e. the loading entity may specify which of the remote interfaces that a pluglet implements should be accessible by which clients. Remote proxies obtained by clients will implement only those remote interfaces that were exposed to them; in fact, clients may not even be aware of the existence of methods that they are not authorized to invoke.

Therefore, there are two types of the policies effective within an execution context: the global,

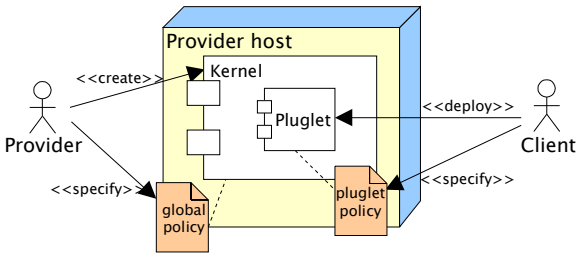


Figure 10: Global and Pluglet Policies

low level policy specified by the kernel provider who explicitly grants permissions to access shared resources, and a pluglet-related policy that determines which interfaces of the specific pluglet should be accessible to particular entities. Figure 10 depicts actors responsible for supplying these policies and shows which components they are tied to.

It is a kernel provider who knows and decides which resources are granted, to whom, and when, so it is up to the provider to specify the global policy. The provider explicitly defines what particular actions (container-specific and/or pluglet-specific) the code executed within a hosting environment may perform. Each and every interaction with the kernel or the hosted pluglet through their remotely accessible interfaces that implies any security sensitive actions (e.g. file system access, opening a socket), is bound to the terms of the policy specified by the kernel provider, and is invoked on behalf of the particular authenticated entity that triggered it. This gives the kernel provider full control over actions that the code loaded into the provider's kernel is permitted to perform.

We have extended the standard Java policy semantics [23] by adding a time-based constraint, which we believe is a crucial feature that runtime environments like H2O absolutely require. At present a kernel provider may grant particular permissions based on the following conditions (as shown in Fig. 11): where the executing code comes from (`codebase` attribute), who it has been signed by (`signedBy`), who invokes it (`principal`) and when it is allowed to be run (`valid`). H2O achieves these controls by substituting the default JRE policy provider with our own implementation that is responsible for reading a policy file from a specific location (as an XML file or any other storage system/format) by means of pluggable storage access modules, as well as for deciding if a particular permission has been granted. Two storage access modules have been developed: an XML policy file format reader and extended policy file format reader (corresponding to examples shown in Figures 11 and 12 respectively).

It is a pluglet deployer, as opposed to a kernel provider, that knows and controls particular interfaces that a pluglet exposes; further, the pluglet deployer is, strictly speaking, the entity that provides the *service* to end-users or clients, albeit using the *resources* belonging to the kernel provider. Therefore it is up to the pluglet deployer either to explicitly specify which interfaces should be accessible to which entities, or omit such information, thus allowing any entity to access any pluglet interface. Needless to say, any action that is allowed by the pluglet policy remains subject to be checked against the global policy.

To ensure privacy and integrity, the Java Secure Socket Extension facilities are used (if requested by the client) with RMI protocols. We are also investigating the applicability of IPSec [12] and PPTP [20] for a layer of VPN-style communication between DVM-enabling components.

```
<?xml version="1.0"?>
<!DOCTYPE policy SYSTEM "XMLPolicy.dtd">

<policy>
   <keystore url="http://localhost/.keystore" type="jks"/>

   <grant codebase="file://java/dev/jaas/Policy/exampleActions.jar" signedBy="duke">
      <valid from="10/9/2002" to="11/8/2003" pattern="MTW:*"/>
      <principal classname="edu.emory.mathcs.security.UserGroup" name="users"/>
      <permission classname="edu.emory.mathcs.security.Permission" target="tgt1" actions="exec"/>
      <permission classname="edu.emory.mathcs.security.Permission" target="tgt1" actions="read"/>
   </grant>

   <grant codebase="file://java/dev/jaas/Policy/*">
      <principal classname="edu.emory.mathcs.security.UserGroup" name="administrators"/>
      <principal classname="edu.emory.mathcs.security.User" name="joe"/>
      <permission classname="edu.emory.mathcs.security.Permission" target="tgt2" actions="read"/>
      <permission classname="edu.emory.mathcs.security.Permission" target="tgt2"/>
   </grant>

   <grant codebase="http://some.trusted.host.net/classes/*" signedBy="trustedPlugletDeployer">
      <valid from="10/25/2002" to="11/25/2002" pattern="*:8.00-9.00;*:10.00-12.00"/>
      <principal classname="edu.emory.mathcs.security.User"/>
      <permission classname="edu.emory.mathcs.security.Permission" target="tgt1" actions="exec"/>
   </grant>
</policy>
```

Figure 11: Example XML Policy File

```
keystore "http://localhost/.keystore", "jks";

grant codeBase "file://java/dev/jaas/Policy/exampleActions.jar",
      signedBy "duke",
      valid from "10/9/2002" to "11/8/2003" pattern "MTW:*",
      principal edu.emory.mathcs.security.UserGroup "users" {
   permission edu.emory.mathcs.security.Permission "tgt1", "exec";
   permission edu.emory.mathcs.security.Permission "tgt1", "read";
};

grant codeBase "file://java/dev/jaas/Policy/*",
      principal edu.emory.mathcs.security.UserGroup "administrators",
      principal edu.emory.mathcs.security.User "joe" {
   permission edu.emory.mathcs.security.Permission "tgt2", "read";
   permission edu.emory.mathcs.security.Permission "tgt2";
};

grant codebase "http://some.trusted.host.net/classes/*",
      signedBy "trustedPlugletDeployer",
      valid from "10/25/2002" to "11/25/2002" pattern "*:8.00-9.00;*:10.00-12.00",
      principal edu.emory.mathcs.security.User * {
   permission edu.emory.mathcs.security.Permission "tgt1", "exec";
};
```

Figure 12: Example Extended Policy File

4 GUI

In operational terms, providers not only wish to maintain full control over their resources, but also desire effective tools to monitor and manage the resources that they share. Therefore, having a solid, reliable, and user-friendly console to manage processes running in a distributed environment is a very important issue. For this reason, H2O includes a Java tool (termed the H2O GUI or simply "GUI") to help users manage their H2O kernels and loaded pluglets from a single, possibly remote, location. The GUI gives a kernel provider the ability to remotely start or stop a kernel process and to manage kernel state. The tool also incorporates the ability to manage kernel access policies in terms of user accounts and user privileges. Moreover, the same uniform GUI interface may be used not only by kernel providers but also pluglet deployers (e.g. third-party resellers of a value-added provider's resource to clients) as well as by end-clients. For these tool users, the GUI provides useful functionality such as a control panel for loading pluglets, searching for previously loaded pluglets, aggregate pluglet management and application progress monitoring.

To facilitate operational convenience, stable data concerning a given provider's resources may be maintained on stable storage in the form of a "profile" file. The profile contains key data concerning a provider's kernels and is maintained in XML-format to enhance interoperability. Given that the GUI is able to access the profile (either locally or remotely), this mechanism facilitates control over routinely shared resources in a manner similar to "roaming profiles" supported by certain Web browsers. Profiles contain a collection of host entries, each containing both static and, after a kernel is instantiated, dynamic information about a host and the H2O kernel on that host. The latter item, called a kernel reference, is of the paramount importance as it is the only way to contact the kernel. When a kernel is started from within the GUI this reference is automatically stored in the profile. Otherwise, the kernel reference must be inserted into the profile manually (either via a dialog within the GUI, by editing the raw file, or other equivalent method. e.g. via XML tools).

An example of single kernel entry is shown in Figure 13. The `RemoteRef` field contains the aforementioned kernel reference that has been omitted for brevity. The `host` and `kernelName` fields identify a kernel in the profile. However, it is possible to have multiple entries with the same host and kernel names, since each *instantiated* kernel will possess a different reference. The remainder of the fields are used during remote kernel startup: `autostart` tells the GUI whether to automatically start a new kernel when it is not yet instantiated or when the remote reference is uncontactable/invalid. The `method` describes the application that will be used to contact the remote machine (e.g. ssh, rsh, etc.) and `parameters` field supplies all additional data for this command. The last field, `command`, contains a path to the H2O kernel startup script on the remote

```
<hostEntry>
   <RemoteRef ... />
   <host>vector.mathcs.emory.edu</host>
   <kernelName>my_red_kernel</kernelName>
   <autostart>true</autostart>
   <method>ssh</method>
   <parameters>yrd@</parameters>
   <command>/h2o/start</command>
</hostEntry>
```

Figure 13: Profile entry

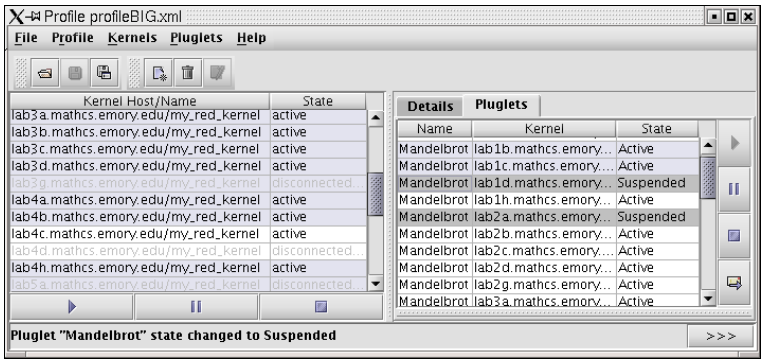


Figure 14: GUI main widow

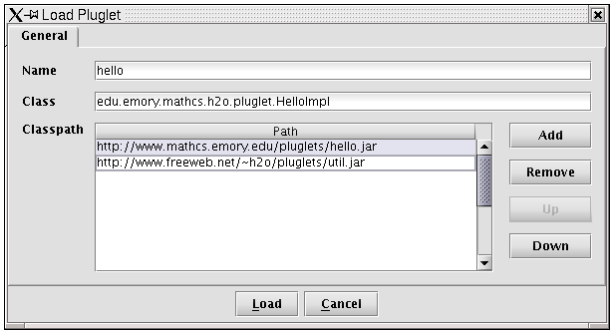


Figure 15: Load pluglet dialog

host.

The GUI has two operating windows. The first is a small frame that contains summary information such as the number of kernels in the profile, the number of active kernels, and the number of pluglets loaded into all kernels. This window will also contain some high level system monitoring data e.g. aggregated workload and resource usage levels for all kernels, connectivity indicators, and other information. It also encompasses controls to shutdown all kernels (i.e. the entire profile) as well as suspend or resume them. The second window, shown in Figure 14, provides the user with a number of new options. Controls from the smaller window continue to be available but separate access to individual kernels is also provided. The left panel displays kernel entries. Either a mouse click or the menu permits access to the individual kernel options. After selecting a kernel, the pluglets loaded into it are shown in the second panel. These may be suspended, reactivated, or terminated.

Selection of one or more contacted kernels enables the "load pluglet" option. A dialog into which the user types in the pluglet loading data is shown in Figure 15. Pluglet loading information may also be provided in the form of a descriptor file which would usually be created by the pluglet provider. All actions that take place within the GUI are logged and may be seen at any time by clicking the "**>>>**" button placed in the bottom right corner of the GUI.

5 Preliminary Experiences

We have developed a prototype implementation of the H2O kernel as described above, along with a set of simple pluglets that allow us to test the viability of the ideas presented. The prototype is fully functional except for final implementation of some of the security features whose design is described in Section 3.3. A rich component library developed in the context of a previous project [17] is currently being migrated to H2O.

One of the early code examples that was developed for H2O is a simple task-farm computing suite. This suite provides abstract middleware for implementing farmer-worker applications, taking care of worker pool management, task rescheduling in the event of worker failure or delay, and optimized worker assignment based on past performance. These features simplify application development, as the farmer responsibility is limited to supplying tasks and receiving results; the middleware guarantees reliability and hides worker assignment. The suite has been used in the H2O demo application that visualizes the Mandelbrot set. The demo enables user to interactively change point of view at the fractal and the scale of the image, and those modifications are visualized instantly – e.g. as the user is dragging a slider bar. To achieve a high level of interaction, the image is computed iteratively with gradually increasing resolution. In every iteration, image is split into regions that are independently computed by available workers. Computational algorithm is entirely captured in the H2O Mandelbrot pluglet loaded in worker kernels. We have successfully run that demo using over 40 general purpose SUN Sparc workstations, geographically distant from the user interface front-end machine.

Although trivial as a distributed computing application, this example demonstrates suitability of the H2O framework for harnessing geographically distributed computational resources in a manner similar to the SETI@home [21] approach. However, this adaptation of the H2O model offers two significant advantages. First, it simplifies development, providing aid in managing worker pools and performing communication. More importantly, it supports dynamic coupling, relieving participating resource providers from installing application-specific modules. The only responsibility of the resource provider is to adjust her pluglet-loading authorization policy (e.g. allowing entities carrying valid SETI@home certificate to load and access pluglets), and to notify interested entities about the willingness to participate in resource sharing (e.g. via the SETI@home Web page). Among other benefits, it allows the client (like SETI@home in this case) to efficiently hot-swap or upgrade distributed application modules.

6 Related Work

The H2O approach to metacomputing emphasizes a lightweight, self-organizing, minimal-state infrastructure as a means of flexible resource sharing via components and the service paradigm. Similar attributes have been explored in several other projects, including ICENI [7], CoG [29], and Harness [17]. Soon after the H2O project was launched, the Open Grid Services Architecture (OGSA) initiative was announced. OGSA integrates grid computing technologies (including the Globus toolkit [6]) with Web Service technologies, and introduces the concept of a Grid Service. There are many similarities, but also substantial differences, between the OGSA and H2O models. OGSA adopts WSDL to describe metacomputing services, whereas H2O permits its usage without mandating it; we believe that in private clusters and "personal area metacomputing", the additional overheads introduced by WSDL may be bypassed. The notion of Transient Service Instances in

OGSA corresponds almost exactly to pluglets in H2O, although there are fundamental differences in their deployment and access, and subtle differences in their discovery and lookup. Both frameworks maintain transparency between local and remote entities, and both support service querying capabilities.

The major differences between OGSA and H2O result from the latter framework's inherent concept of a component container with well-defined semantics and capabilities. OGSA aims to encompass diverse service hosting environments, ranging from those as simple as the fork/exec system call to sophisticated enterprise containers like J2EE or .NET. As a consequence, the focus in OGSA is on interactions between services and clients, leaving most aspects of interactions between services and the container underspecified. For instance, to address the issue of service deployment, OGSA proposes the notion of factories – that requires *providers* to supply classes of services although clients may *instantiate* them. In contrast, H2O components may be deployed not only by providers but by any authorized parties. The container serves as a generalized factory capable of instantiating any service in a secure and controlled manner, even if its exact type is not known to the provider. Furthermore, the currently available OGSA implementation (contained in the Globus Toolkit v3 technology preview) is strongly dependent on the SOAP protocol, unsuitable for high performance communication [2]. As a result, the usefulness of OGSA service interfaces is limited to management purposes, whereas the actual application components are forced to use alternative, application-specific communication channels. In contrast, H2O is based on a flexible, multiprotocol communication substrate and provides application developers with a well-defined and well-established communication model.

7 Summary

In this paper, we have outlined the architecture and design of the H2O framework, a lightweight and self-organizing infrastructure for distributed metacomputing. By reducing state, and permitting services to be uploaded and configured by clients, flexible, dynamic, and robust resource sharing is enabled, without compromising provider safety and control. Our early experiences with the model and a prototype implementation are encouraging and we believe that this complementary approach to computational grids may contribute viable strategies for next generation distributed metacomputing.

Acknowledgements

This work was supported in part by NSF grant ACI-0220183 and DoE grant DE-FG02-02ER25537.

References

[1] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Supercomputing 2001 Conference*, Denver, Colorado, USA, November 10-16 2001. Available at `http://www.cactuscode.org/Papers/GordonBell_2001.ps.gz`.

[2] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High*

Performance Distributed Computing, pages 246–254, Edinburgh, Scotland, 2002. Available at http://www.extreme.indiana.edu/~mgovinda/research/papers/soap-hpdc2002.pdf.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/.

[4] K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999. Available at http://citeseer.nj.nec.com/falkner99implementing.html.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The Intl Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[6] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of Supercomputer Applications*, 15(3), 2001.

[7] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of component-based applications within a grid environment. In *Supercomputing 2001 Conference*, Denver, Colorado, USA, Nov. 2001. Available at http://www-icpc.doc.ic.ac.uk/components/papers/sc2001.ps.gz.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[9] Grid physics network. http://www.griphyn.org.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[11] IETF. Internet X.509 public key infrastructure proxy certificate profile. http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-02.txt.

[12] IETF. IP Security Protocol (IPSec). http://www.ietf.cnri.reston.va.us/html.charters/ipsec-charter.html.

[13] Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/.

[14] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: Multiprotocol RMI framework for Java. In *Java Parallel Distributed Computing Workshop*, Nice, France, 2002. (submitted).

[15] Microsoft Corporation. .NET framework: Product information overview. http://msdn.microsoft.com/netframework/prodinfo/.

[16] Microsoft, VeriSign, and IBM. Web Services Security (WS-Security). http://www-106.ibm.com/developerworks/library/ws-secure/.

[17] M. Migliardi and V. Sunderam. The Harness metacomputing framework. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, USA, March 22-24 1999. Available at `http://www.mathcs.emory.edu/harness/PAPERS/pp99.ps.gz`.

[18] NASA. Information Power Grid. `http://www.ipg.nasa.gov/`.

[19] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. Grids: Harnessing geographically-separated resources in a multi-organisational context. In *15th Annual International Symposium on High Performance Computing Systems and Applications*, June 2001. Available at `http://legion.virginia.edu/papers/HPCS01.pdf`.

[20] B. Schneier and Mudge. Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). In *Proceedings of the 5th ACM Conference on Communications and Computer Security*, Nov. 1998. Available at `http://www.counterpane.com/pptp-paper.html`.

[21] SETI@home: Search for extraterrestrial intelligence at home. `http://setiathome.ssl.berkeley.edu/`.

[22] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based interoperable RMI system: SoapRMI C++/Java. In *Proceedings of Parallel and Distributed Processing Techniques and Applications Conference*, Las Vegas, NV, USA, June 2001. Available at `http://www.extreme.indiana.edu/soap/rmi/design/`.

[23] Sun Microsystems. Default policy implementation and policy file syntax. `http://java.sun.com/j2se/1.4.1/docs/guide/security/PolicyFiles.html`.

[24] Sun Microsystems. Java 2 Platform, Enterprise Edition overview. `http://java.sun.com/j2ee/overview.html`.

[25] Sun Microsystems. Java Authentication and Authorization Service (JAAS). `http://java.sun.com/products/jaas/`.

[26] Sun Microsystems. Pluggable Authentication Modules (PAM). `http://www.sun.com/software/solaris/pam/`.

[27] United Devices. Project: Cancer research. `http://members.ud.com/projects/cancer/`.

[28] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, Jan-Mar 1999. Available at `http://www.cs.vu.nl/~steen/globe/`.

[29] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001. Available at `http://www.mcs.anl.gov/~laszewsk/papers/cog-cpe-final.pdf`.

[30] WebServices.Org. `http://www.webservices.org/`.