Specification of the *2K* Services (A draft for internal use only)

March 2, 1999

1 Introduction

The *2K* operating system presents itself to application developers as a collection of distributed services whose interfaces are defined using OMG IDL. This document provides a brief description of the most important services and includes the complete IDL definition for each of them.

Figure 1 shows the different layers that form *2K*. The objective of this document is to provide *2K* team members with a better understanding of the system upper layers and help us on the integration of the multiple subprojects. Documentation about the lower layers can be found in the *2K* home page. For a more detailed description of each of the services presented in this document, refer to the service specific documentation available in the *2K* `Internal` page or contact the student responsible for the subproject.

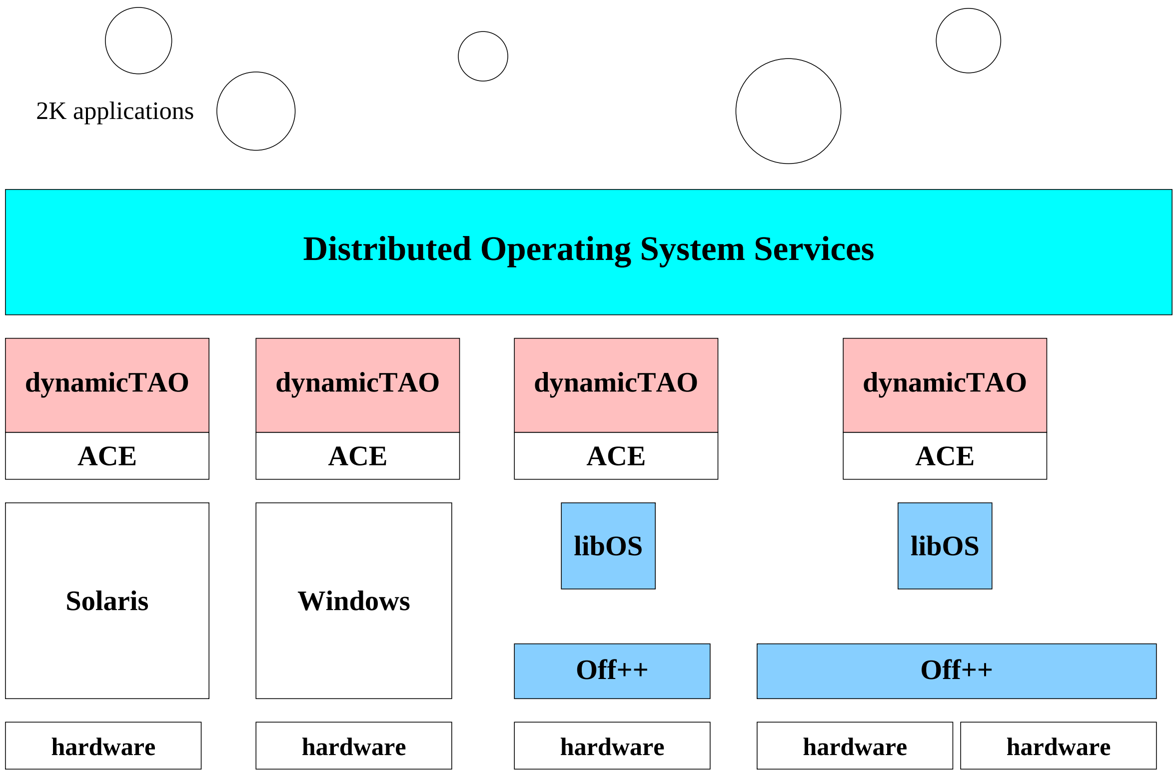

Figure 1: Overview of the *2K* layered architecture

This document describes the following *2K* services:

- Naming Service
- Dynamic Configuration Service
- Security Service
- Environment Service *(needs clarification of some issues)*
- Adaptive Distributed Multimedia *(almost empty)*
- Quality Of Service Management *(needs work)*
- Resource Management Service *(needs work)*
- Dependency Management Service

2 Naming Service

The *2K* Naming Service is basically an extension of the CORBA standard name service, implementing exactly the same interface. In addition to the facilities described in the standard, our Naming Service offers *junctions*, a mechanism for bridging to other naming systems (such as the ones implemented by POSIX file servers) and *clerks*, local modules that contact remote name servers in behalf of the user applications and perform optimizations such as caching. The very existence of junctions and clerks is transparent to common users that simply see a standard CORBA naming service. *2K* administrators, on the other hand, must configure junctions and clerks to provide the desired service.

The *2K* Naming Service is based on the TAO implementation which implements a multicast-based discovery algorithm for locating name servers in the local network so that clients do not need to know in which machine the name server is located. Alternatively, the address of a particular name server can be provided in the command-line options for any CORBA client or server.

Dulcineia is maintaining a default *2K* Persistent Name Server on the SRG network that should be shared by the whole *2K* team.

As shown in figure 2, *2K* will have a global name space where references to *2K* clusters and references to global services will be stored. In the name space depicted in this figure, the name `/uiuc/dcs/MONET` refers to the *2K* cluster called MONET, while the name `EnvironmentService` refers to the Environment Service. One can expect that the `/uiuc/dcs/MONET` will contain the same value all over the globe, i.e., a reference to the MONET cluster. The `EnvironmentService` name, on the other hand, may contain a reference to the local instance of the Environment Service and, thus, vary from location to location.

Figure 3 shows an example of how the name space of a *2K* cluster may look like. Ovals represent *contexts* (equivalent to UNIX directories), rectangles with round corners represent junctions, and rectangles with straight corners represent bindings to CORBA objects. The figure contains an NFS junction, and subtrees for *2K* users, local services, machines in the cluster. Within the SRG cluster, the relative name `services/multimedia/VoD/lectures/cs314/3`, for example, is mapped to a CORBA object encapsulating a video of the third CS314 class.

More details about the *2K* Naming Service design and implementation can be found in Zia's Masters Thesis in the *2K* home page.

For convenience, the CORBA naming service interface is presented bellow.

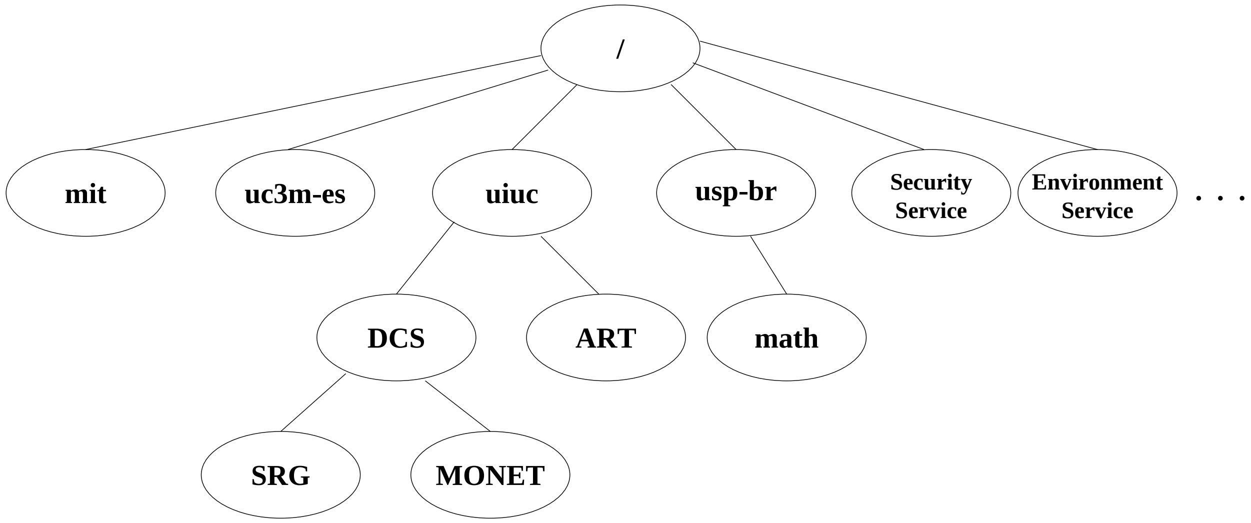

Figure 2: Global *2K* name space

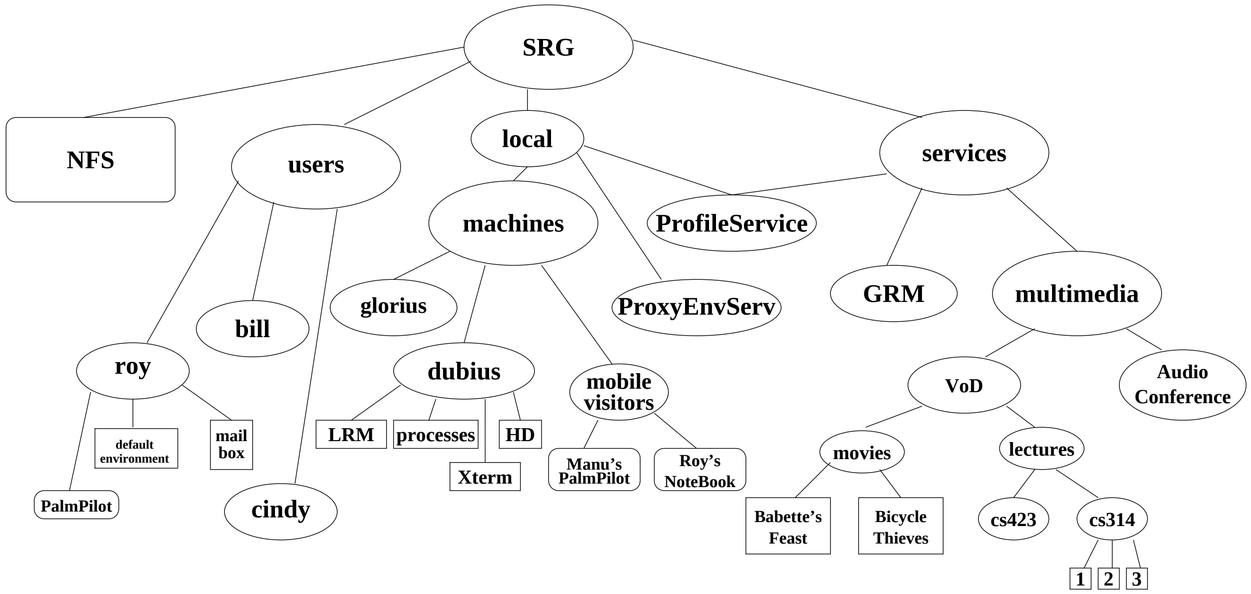

Figure 3: Name Space for the SRG *2K* cluster

```
module CosNaming
{
  typedef string Istring;
  struct NameComponent
  {
    Istring id;
    Istring kind;
  };

  typedef sequence <NameComponent> Name;

  enum BindingType { nobject, ncontext };

  struct Binding
  {
    Name     binding_name;
    BindingType binding_type;
  };

  typedef sequence <Binding> BindingList;

  interface BindingIterator;  // Forward declaration.

  interface NamingContext
    {
      enum NotFoundReason { missing_node, not_context,not_object };
      exception NotFound
       {
         NotFoundReason why;
         Name rest_of_name;
       };
      exception CannotProceed
       {
         NamingContext cxt;
         Name rest_of_name;
       };
      exception InvalidName {};
      exception AlreadyBound {};
      exception NotEmpty {};

      void bind (in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

      void rebind (in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName);

      void bind_context (in Name n, in NamingContext nc)
```

```
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

      void rebind_context (in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName);

      Object resolve (in Name n)
        raises(NotFound, CannotProceed, InvalidName);

      void unbind (in Name n)
        raises(NotFound, CannotProceed, InvalidName);

      NamingContext new_context ();

      NamingContext bind_new_context (in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);

      void destroy ()
        raises (NotEmpty);

      void list (in unsigned long how_many,
                 out BindingList bl, out BindingIterator bi);
    };

  interface BindingIterator
    {
      boolean next_one (out Binding b);

      boolean next_n (in unsigned long how_many,
                      out BindingList bl);
      void destroy ();
    };
};
```

3 DynamicConfiguration Service

Contact: Manuel.

The *2K* reflective ORB, *dynamicTAO*, uses the *Strategy* design pattern to separate different aspects of the ORB internal engine. A configuration file is used to specify the default strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded.

In addition, *2K* offers the *Dynamic Configuration Service* which provides dynamic reconfiguration. This service can be used both for reconfiguring the ORB internal implementation (selecting different strategies) and for reconfiguring CORBA applications running on top of the ORB.

Each ORB maintains a repository of component implementations in its local file system. The Configuration Service provides operations for uploading new implementations to the ORB repository, for loading and unloading implementations into the runtime system, for browsing and inspecting the (system and application) components in the ORB domain, and for changing the bindings between running components.

The Dynamic Configuration Service can be accessed through (1) a telnet connection using the Distributed Configuration Protocol (DCP), (2) a GUI implementing the DCP or (3) a CORBA interface. A detailed explanation of each of the available operations can be found in the DCP specification available at `http://choices.cs.uiuc.edu/2k/DCP`. The IDL interface follows:

```
interface DynamicConfigurator
{
 typedef sequence<string> stringList;

 exception SERVICE_ERROR {};

 stringList list_categories()
     raises(SERVICE_ERROR);
 stringList list_implementations(in string categoryName)
     raises(SERVICE_ERROR);
 stringList list_loaded_implementations ();
 string get_impl_info (in string implName)
     raises(SERVICE_ERROR);
 stringList list_domain_components ();
 stringList list_hooks (in string componentName)
     raises(SERVICE_ERROR);
 string get_comp_info (in string componentName)
     raises (SERVICE_ERROR);
 void load_implementation (in string categoryName,
                           in string impName,
                           in string params)
     raises(SERVICE_ERROR);
 void hook_implementation (in string impName,
                           in string componentName,
                           in string hookName)
     raises(SERVICE_ERROR);
 void suspend_implementation (in string impName)
     raises(SERVICE_ERROR);
 void resume_implementation (in string impName)
     raises(SERVICE_ERROR);
 void remove_implementation (in string impName)
     raises(SERVICE_ERROR);
 void configure_implementation (in string impName)
     raises(SERVICE_ERROR);
 string get_hooked_comp (in string componentName, in string hookName)
     raises (SERVICE_ERROR);

 // The following operations are only available in the DCP interface.
 // void upload_implementation (in string request);
 // void delete_implementation (in string serviceName, in string impName);
};
```

4 Security Service

Contact: Ping.

The *dynamicTAO* ORB was extended by Manuel and Ping to include the mechanisms needed for security (the original TAO does not support it).

The client side was modified too add *principal* information into the IIOP message. In the server side, an "interceptor" was added to the operation `TAO_Server_Connection_Handler::handle_message()` to call the `check()` operation on the server security strategy.

Using the DynamicConfigurator interface (see section 3) or the configuration telnet interface, one can load different security strategies dynamically. At the present moment, the only implemented security strategies are the Cherubim_Strategy and a Null_Strategy that always grants access. If a security strategy is not defined, the ORB always grants access. Once a security strategy is specified, it is valid for all objects in that process. Obviously, different processes can use different security strategies implementing different mechanisms.

Using Cherubim tools, the user of the Security System can specify access permissions for each operation in each object handled by a certain CORBA server.

The Cherubim_Strategy depends on a Java virtual machine to run. Thus, it was integrated to *dynamicTAO* as a loadable strategy called `Interpreting_Strategy`. This JVM is not tied to the Security System and can be used by any other module or CORBA application to execute Java bytecode within *dynamicTAO*.

At the current moment we do not have an implementation of an strategy for encryption, but it should be done sometime in the future. Ping is currently working on a faster implementation of the Security System in which the access permissions will be cached in a *Reference Monitor*. This monitor will be implemented in C++ so it will be no longer necessary to execute Java code every time a security check is performed.

Security checks are performed in a completely transparent way. Once the DynamicConfigurator is used to load a security strategy, the security checks are performed automatically without intervention from the user or application.

By the end of February/99 we should have an idea of the interfaces exported by the new, optimized Security Service based on the Reference Monitor. By now, we present the currently available Security interface which builds heavily on the Cherubim framework.

The Cherubim interface is invoked to specify active capabilities which are stored in the local *AC Manager*.

```
interface ACManager{
    //update capability
    void update(in string obj_name, in string user_name, in string cap);

    //check if the request can be allowed or not
    void enforce(in string obj_name, in string user_name, in string cap,
                 in ACManager acm);

    //change capability attributes
    void change(in string  obj_name, in string cap);

    //just for test
    void test();
};
```

```
interface PolicyServer{
   //Register Principal's Capability to Policy Server.
   void register(in string obj_name,
                 in string user_name,
                 in string cap,
                 in ACManager acm);
   //ACM can obtain AC from policyServer
   string obtain(in string obj_name,
                 in string user_name,
                 in ACManager acm);
};
```

Active capabilities are Java objects (not CORBA objects) with the following interface.

```
public class ActiveCapability implements Serializable {
  protected String signer, algorithm;
  private byte[] signature;

  //sign the AC using Algorithm alg by Signer sig
  public synchronized void sign (String sig, String alg) {}

  //verify the signature signed by "sig"
  public synchronized boolean verify () {}

  public void Export(OutputStream out) {}

  public void Import(InputStream in) {}

  private void writeObject(ObjectOutputStream out) throws IOException {}

  public void revoke() {}               // not implemented
  public ActiveCapability delegate() {} // not implemented
}
```

For details on the use of the cherubim security strategy, contact Ping Liu. As new security strategies are implemented we may publish new interfaces for customizing a particular security strategy.

Open Problem : Can we have a single Security Service interface that would work with multiple underlying security mechanisms? Can different security mechanisms interoperate?

5 Environment Service

Contact: Dulcineia.

The environment service is responsible for managing groups of components as a collective entity and for helping the components interface with the *2K* operating system.

An environment is a container of `Component`s and `Device`s and is created with the operation `Environment_Service::create_environment`. A client creating a new environment must specify a `Profile` describing the initial, default components and devices as well as the (hardware) resource requirements for that environment. Optionally, the client may also specify the location in which the environment will be primarily located.

Typically, users will have default profiles that can be customized depending upon their location, role, and the applications that he or she intends to execute.

When a environment is created, the system must make sure that the resources specified in the profile are available. This is accomplished by contacting the Resource Management Service (see section 8). Then, the environment service iteratively brings to the environment all the components and devices that it requires. The specification of components in a profile (`CompSpec`) may contain one of three options: (1) a CORBA reference, (2) a naming service name or (3) a list of properties to be passed to the Trader Service in order to obtain a component name.

As the components are brought to the environment (either by reusing existing CORBA objects or creating new ones), the `Component::dependencies` object is updated to represent the runtime dependencies between the components in the environment and other (possibly external) components. `Component::dependencies` is an instance of the ComponentConfigurator defined in section 9.

When an environment is terminated, with the `Environment_Service::close_environment` operation, it is possible to specify the location of a persistent repository in which the environment state will be saved.

Points needing further clarification:

- Where user/principal `Profile`s are stored and where the `Component`s objects are stored. How they are retrieved?
- How new components are dynamically loaded (probably from a group of implementation repositories).
- More details about the interaction with the Resource Management and QoS Management services.

```
#include "CosPropertyService.idl"
#include "CosNaming.idl"
#include "Configuration.idl"


module twokEnvironmentService{
  interface Device;
  interface Component;
  interface Environment;
  typedef sequence<Component> ListOfComponents;
  typedef sequence<Device> ListOfDevices;
```

```
typedef sequence<Environment> ListOfEnvironments;
enum SpecDiscriminant {CompName, CompPropertyList, CompReference};
union CompSpec switch (SpecDiscriminant)
{
    case CompName: CosNaming::Name comp_name;
    case CompPropertyList: CosPropertyService::PropertySetDef comp_property_list;
    case CompReference: Object comp_reference;
};
typedef sequence<CompSpec> ListOfCompSpecs;
typedef CosPropertyService::PropertyDef Desc;
typedef sequence<Desc> ListOfDescs;

typedef string ResourceName;
struct Resource{
  ResourceName resource_name;
  float resource_value;
};
typedef sequence<Resource> ListOfResources;

enum CloseOperationType {PERSISTENT, NON_PERSISTENT};

exception InsufficientResources {};
exception InvalidLocation {};
exception UnknownObject {};
exception BadExecution {};
exception BadInstallation{};

struct Profile {
  ListOfResources default_resources;
  ListOfDescs default_devices;
  ListOfCompSpecs default_components;
};

interface Environment {
  attribute Profile profile;
  attribute ListOfComponents components;
  attribute ListOfDevices devices;

  Component add_component(in CompSpec comp_spec)
    raises (InsufficientResources, UnknownObject);
  Component load_component(in CompSpec comp_spec)
    raises (InsufficientResources, UnknownObject);
  void remove_component(in Component comp_ref)
    raises (UnknownObject);
  Device add_device(in Desc dev_desc)
    raises (UnknownObject);
  void remove_device(in Device dev_ref)
```

```
      raises (UnknownObject);
  };

  interface Component {
    attribute Desc description;
    attribute ListOfCompSpecs component_prerequisites;
    attribute ListOfCompSpecs device_prerequisistes;
    attribute ListOfResources resources_prerequisites;
    attribute Configuration::ComponentConfigurator dependencies;
    // Uses the Component Configurator hooks to store the IORS of the
    // the dependencies
  };

  interface Device {
    attribute Desc description;
    attribute ListOfResources resources_prerequisites;
    attribute Configuration::ComponentConfigurator dependencies;
  };

  interface Environment_Service {
    attribute ListOfEnvironments environments;
    Environment create_environment(in Profile profile, in CosNaming::Name location)
       raises(InsufficientResources, InvalidLocation, UnknownObject);
    void close_environment(in Environment env_ref, in CloseOperationType close_Op,
                           in CosNaming::Name persistent_store)
       raises (InsufficientResources, UnknownObject);
  };
};
```

6 Quality Of Service Management

Contact: Dao.

This module defines a group of interfaces for specifying QoS parameters required or exported by clients and servers. `Translator`s are used to map QoS data from one domain to another (e.g., from application-level parameters such as frame rate to kernel-level parameters such as CPU and memory utilization). `Comparator`s check whether the QoS offered by a service at a particular time is enough to satisfy a client's QoS requirements.

Finally, a `QoSProxy` acts as a registry for QoS-aware services that can be used by servers to publicly announce their services and by clients to locate and request services.

Note: This interface needs to change a bit in the near future to fix some problems.

```
interface ServicePropertyTemplate : CosPropertyService::PropertySetDef
{
  // Defines a set of QoS parameters for a particular service.
} ;

interface PropertyTemplate
{
        void initialize(inout ServicePropertyTemplate qos_list) ;
} ;

interface QoSObject
{
        long start_service() ;
        void terminate_service() ;
} ;

interface Translator
{
  exception IllegalArgument {} ;

  void translate(inout ServicePropertyTemplate qos_list)
    raises(IllegalArgument) ;
};

interface Comparator
{
  boolean Is_Satisfied(in ServicePropertyTemplate client_qos_list,
                       in ServicePropertyTemplate service_qos_list);
};

interface QoSProxy
{
 exception RequestedServiceNotFound {} ;
 exception RequestedServiceNotSupported {} ;
 exception NoRequestedServiceName {} ;
```

```
 exception NotEnoughCpu {} ;

 void request_service(in long client_pid, in string service_name,
                      in ServicePropertyTemplate app_qos, in QoSProxy qosProxy,
                      in CosNaming::NamingContext naming_context)
     raises(RequestedServiceNotFound, NotEnoughCpu) ;
 void release_service(in long client_pid, in short original) ;

 boolean is_supported(in string service_name, in long client_pid,
                      inout ServicePropertyTemplate app_qos_plus_comminfo_list,
                      in QoSProxy qosProxy,
                      in CosNaming::NamingContext naming_context)
     raises(RequestedServiceNotSupported) ;
 void register_service(in string service_name,
         in ServicePropertyTemplate service_qos) ;
 void remove_service(in string service_name)
     raises(NoRequestedServiceName) ;
};
```

7 Adaptive Distributed Multimedia

Contact: Dongyan.

This subproject investigates adaptive resource management for **distributed** multimedia applications where bandwidth control is the main concern. Differently from resources like CPU and memory, network bandwidth cannot be managed locally; it is intrinsicly distributed. Thus, we need a mechanism to deal with this specific problem.

Until the present moment, the framework for Adaptive Distributed Multimedia was thought as something disconnected from the *2K* operating system. We will now concentrate on how to integrate this framework into the system. It must be thought as a *2K* subsystem. To do so, three things need to be done:

1. Specify what this service will provide and how it will export its services to *2K* applications. Define the IDL interface.

2. Construct the subsystem internal structure in terms of CORBA components and specify the interactions between them.

3. Specify how the subsystem components will utilize *2K* facilities such as the Naming Service, Security Service, Environment Service, Dependency Management, etc.

.

8 Resource Management Service

Contact: Fabio (by now), Tomonori (in the future? (yamane@uiuc.edu)).

The Resource Management Service collaborates with the QoS Management Service to make the *distributed* hardware resources available to *2K* components. It is based on two major components. The *Local Resource Managers* (LRMs) interfaces with the local operating system to inspect system state and reserve local resources. The *Global Resource Managers* (GRMs) collect and maintain consolidated data about the state of the machines in a *2K* cluster and uses it as a hint to direct client requests. Machines are put together in clusters by system administrators based on criteria such as security domain, machine type, or physical location. Example of clusters are: all the machines in a LAN, all the (heterogeneous) machines of a single user, all the machines in a security domain, or all the machines in one office. GRMs in different clusters collaborate with each other to share resources across *2K* clusters.

A detailed description of the Resource Management protocols can be found in `http://choices.cs.uiuc.edu/2k/Internal/design/ResourceManagement`. We have the code of a prototype of these protocols developed by CS423 students. But, there is still a lot of work to be done to make it flexible and to adapt it to *2K* needs. We need a person to work on that (Tomonori?).

9 Dependency Management Service

Contact: Fabio.

One of the major characteristics of *2K*, and also what differentiates it from other operating systems, is its *architectural-awareness*. Application and system components maintain an explicit representation of its inter-dependencies. This awareness is present in all the layers of the operating system, from the Off++ microkernel up to the user-level CORBA applications. The basic mechanism that reifies inter-component dependencies is the `ComponentConfigurator`. It comes in three flavors: C++ and Java implementations for centralized applications and a CORBA implementation for distributed applications.

The CORBA implementation use IORs to represent inter-component dependencies and to refer to component implementations. CORBA ComponentConfigurator objects are created by calling a remote `Configuration::Factory` or by directly instantiating a new object in the local domain. Applications and subsystems may choose to keep their configurator objects in a single "administrative" node to facilitate management or to keep them distributed over the network, close to the components they manage.

More information about this topic can be found at `http://choices/2k/ComponentConfigurator`. The CORBA interface follows.

```
module Configuration
{
  enum EventType {DELETED, FAILED, RECONFIGURED, REPLACED, MIGRATED, UNKNOWN};
  struct Event
  {
    EventType type;
    string description;
  };
  interface ComponentConfigurator; // forward declaration
  struct DependencySpecification
  {
    string name;
    ComponentConfigurator componentConfig;
  };
  typedef sequence<DependencySpecification> DependencyList;

  interface ComponentConfigurator
    {
      exception invalidArgument{};
      exception ElementExists{};
      exception NotFound{};
      exception HookBusy{};
      exception HookVacant{};

      void addHook          (in string hookName) raises (ElementExists);
      void deleteHook       (in string hookName) raises (NotFound);
      void hook             (in string hookName, in ComponentConfigurator cc)
          raises (NotFound);
      void unhook           (in string hookName) raises (HookVacant, NotFound);
```

```
    void registerClient   (in ComponentConfigurator client,
                           in string hookNameInClient) raises (ElementExists);
    void unregisterClient (in ComponentConfigurator client,
                           in string hookNameInClient) raises (NotFound);
    void eventOnHookedComponent (in ComponentConfigurator hookedComponent,
                                 in Event e, in unsigned short timeToLive);
    void eventOnClient          (in ComponentConfigurator client,
                                 in Event e, in unsigned short timeToLive);
    string name ();
    string info ();
    DependencyList listHooks ();
    ComponentConfigurator getHookedComponent(in string hookName);
    unsigned short numberOfClients ();
    DependencyList listClients ();
    Object implementation ();
    void destroy ();
  };

 interface Factory
  {
    ComponentConfigurator newConfigurator (in string name,
                                           in Object implementation);
    oneway void shutdown ();
  };
};
```