ICAT 2005
December 5-8, Christchurch, New Zealand

Self-Aware Framework for Adaptive Augmented Reality

Eduardo E. Veas[1] Kiyoshi Kiyokawa[1,2] Haruo Takemura[1,2]

Graduate School of Information Science and Technology, Osaka University[1]
Cybermedia Center, Osaka University[2]

Abstract

In the dynamic environments of UAR, software is expected to be modifiable to accommodate for new user needs, expectations, and changing environments. Reflective middleware provides, in a certain way, an open implementation and presents an efficient alternative to deal with highly dynamic environments. The present article describes an implementation of Reflective Middleware based on an object oriented metamodel and its application in the field of AR. Along the way, it introduces support services that deterministically apply to every component, even those yet to be engineered. These services have been defined as Generic Services, the process of creating *generic services* is demonstrated with various examples, showing their advantages as regards to issues of code reduction and flexibility.

Keywords: ubiquitous computing, augmented reality, reflective middleware, dynamic reconfiguration, self-aware architecture, metamodel, generic services, distributed systems

1. Introduction

As already signaled by [14] augmented reality, particularly wearable AR, is beginning to overlap with ubiquitous computing. Ubiquitous Augmented Reality is defined as the intersection of these two fields. In his paper, Mac Williams further introduced challenges for the development of applications for UAR. These include *uncertainty*, given by the need for components with different availability to interact in a changing context with incomplete knowledge of other components. *Ill-definition* of the expected system behavior, produced as a result of combining new and changing technologies, users and specialists from a wide range of fields. And *performance*, imposed by the interactive and immersive nature of mixed reality applications.

As an example, consider a user A entering a room carrying her own computing devices. In the room, another user B engages in an architecturing prototype, using the demo application of section 5. Meanwhile two other users, C and D, play an AR game of chess. User A should receive notification of availability of other applications in her computing devices. She should also be able to engage in collaborative activities, for example with user B, receiving support to load any new components that might be required. This example fits the definition of UAR [14], and the room area can be defined as an *augmented area*. Furthermore, a user E should possibly be virtually navigating the city being constructed by user B, in a remote place, through a completely virtual interface, this would make the application demo an example of a *mixed area* where augmented and virtual interfaces mix. Finally, as user A leaves, she should be able to unload all unnecessary components, and also take all those she found interesting enough.

Our goal is the development of an unobtrusive architecture for UAR. To provide for the above mentioned challenges, such a architecture should include an *adaptable structure* based on an *extensible set of components*, and a common software infrastructure or *middleware* supporting different communication protocols and scalability.

Various standard approaches provide support for distributed systems [11] [2], and some frameworks and toolkits have been based on these approaches. These middlewares often offer component models that support third party development, composition and deployment, and the approaches to handle self-awareness are added by each solution. However, these models are in general offered on top of the middleware platform. As highlighted by research in reflective middleware [8], there are advantages to also exploit component techniques and reflection within the middleware. Key among these advantages are the possibility to adapt to changes in the environment, and to customize the options to fit into wide range of devices. These advantages have been applied, in the presented proposal, to the creation of support services, defined as *generic services*.

In the following, an infrastructure that provides a base layer for development of these changing environments is presented. Insight on related work, and a brief description of the application of reflection in areas related to AR are provided in the following section. In section three the reflective infrastructure serving as foundation for generic services and self-awareness is presented. Example services created on top of the metamodel are introduced in section four. Finally examples and conclusions are presented.

2. Related Work

"A distributed system should easily connect users to resources; it should hide the fact that resources are distributed across a network; it should be open and it should be scalable" [20]. Different approaches at distribution are useful in different situations. When the state of the distributed part is not needed in every host and the processing can be carried out by a single computer, remote invocation can be used. There are situations when the state is needed in every host, for example the graphic models are needed to render, then a replication approach can be used. There are situations when hybrid approaches are recommendable. Various standard approaches provide support for distributed systems CORBA [11], Java RMI [2]. These approaches often require careful planning of the interfaces and how they will be used. Once the components have been properly designed, applications can be developed in different scenarios, always supported by the middleware's services. However, these standards not always provide services for all the communication needs, for example JavaRMI does not support active or passive replication. Decisions on the way interfaces will be used can not be delayed.

Ubiquitous Computing has raised the requirement for flexibility in the software infrastructure layer. Middleware solutions focused in this area propose extending distributed systems, making them aware of their structure. Early approaches were based on keeping an connection map between the components of the architecture [18] [13]. The OpenCOM project uses metamodels to describe the connections between components and the architecture of the given solution [6]. OpenCOMv2 is expected to extend the middleware to provide a multilanguage support while RUNES [7] attempts to extend the above ideas to embedded systems. Reflectivity is argued to be “an efficient way to deal with highly dynamic environments, supporting the development of flexible and adaptive systems and applications” [8]. These approaches are mainly focused on reconfiguration problems.

In Virtual Reality area, approaches aiming at distributed virtual environments, have proposed mechanisms such as distributed scenegraphs [21] [16] [19]. Following the ideas behind designs of scene-graph toolkits ([24] among others), modular approaches, intended for flexible virtual environments [23] [21] identified the need of a minimum reflective infrastructure, to at least recognize types in the modules.

Given the variety of sensors, tracking, input and display devices it deals with, reconfiguration capabilities have been considered by most efforts in Augmented Reality architectures. The Tinmith toolkit [17] focuses on mobile outdoor AR, and uses a data driven, pipelined architecture. Reconfiguration of the system relies on reparsing stored objects to change runtime values. Studierstube is built on top of Open Inventor; applications written as separate modules are loaded dynamically into the runtime framework, decoupling application development from system development. Besides, Studierstube relies on OpenTracker to accommodate tracker data and on a distributed scenegraph [19] for distribution. Developed with focus in UAR, Dwarf [4] is based on the idea of distributed services. Each network node runs a service manager, which keeps information about local services such as what a service offers to others, its “abilities”, and what it requires to properly function, its “needs”. This structural information can be used to reconfigure the system on the fly through a Petri net GUI, and can be regarded as reflective information about the services, providing enough structural awareness to be able to change the software structure at runtime. One difference in this approach is it does not provide enough reflective means to manipulate unknown interfaces (other than supported by the middleware), thus lacking support for Generic Service. On the other hand DWARF relies on CORBA to realize distributed services and to handle communications, adding the reflective information for reconfiguration. This not only adds up to the layering of the system, but it also gives up advantages of using component based techniques and reflection within the middleware platform.

Our approach relies on a metamodel with structural and also behavioral characteristics, which provides manipulation capabilities in addition to structural information. Similar metamodels have been supported by programming languages, particularly interpreted languages [1], Python. However, they have seldom been used to create generic services, like remote objects, and are often provided as one more feature of the language. In Java RMI for example, the designer is forced to inherit from particular interfaces, that is to also plan whether objects should be remote invocable. Such decisions of how a certain component would be used can be deferred until deployment time or even runtime by relying on the metamodel.

3. Base architecture: metamodel

Self-aware architectures contain runtime information that allows them to *reflect* upon themselves and adapt to changes in the environment. “A component implements one or more *interfaces* that are imposed upon it” [3]. Interfaces, in object oriented architecture, are described by classes whose features define their behavior [15]. A class can contain creation features, attribute features and method features.

The metamodel (Figure 1) provides abstract interfaces to the above mentioned entities. It also includes a base interface (IMetaObj) for components that support the metamodel. A component supporting the metamodel must, minimally, provide a way to retrieve an instance of the *IClass* interface, from which instances of the rest of the metamodel can be acquired. In general, it is only necessary to provide a handle to the function used to create the instances of the required interfaces (IClass, IMethod, etc.). To facilitate such a task, macros that instantiate derived template classes are provided. Therefore, to export the metamodel for the class in Table 1, a declaration of the form in Table 2 is used.

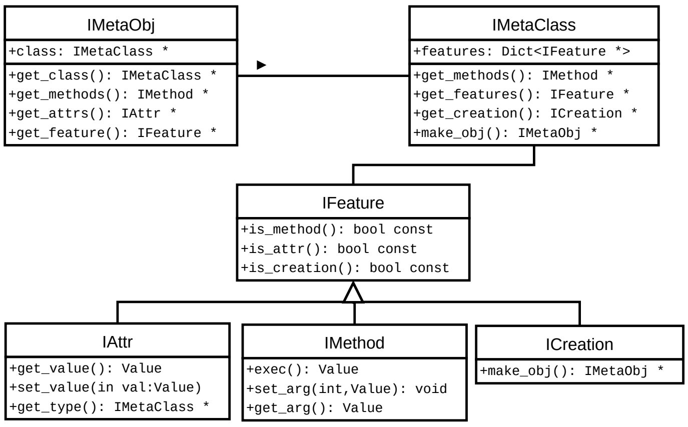

IMetaObj
+class: IMetaClass *
+get_class(): IMetaClass *
+get_methods(): IMethod *
+get_attrs(): IAttr *
+get_feature(): IFeature *
IMetaClass
+features: Dict<IFeature *>
+get_methods(): IMethod *
+get_features(): IFeature *
+get_creation(): ICreation *
+make_obj(): IMetaObj *
IFeature
+is_method(): bool const
+is_attr(): bool const
+is_creation(): bool const
IAttr
+get_value(): Value
+set_value(in val:Value)
+get_type(): IMetaClass *
IMethod
+exec(): Value
+set_arg(int,Value): void
+get_arg(): Value
ICreation
+make_obj(): IMetaObj *
Figure 1: Base interfaces of the metamodel.

Table 1: ARToolkit tracking class.

```
class ARtrack
 public:
  ARtrack();
  virtual ˜ARtrack();
  void set_config_file(String file);
  String get_config_file();
  void set_fitting_mode(int fm);
  int get_fitting_mode();
  void set_imageproc_mode(int ipm);
  int get_imageproc_mode();
  void add_marker(ARmarker * marker);
  int load_pattern(String filename);
  void detect_markers();
  void select_optimal();
  ARmarker * get_marker(String name);
  void resolve_transform();
```

The macros mimic a class declaration, and they could be extracted from a header file by a preprocessor (i.e. idl compiler). However, it is not necessary for a certain component to use the provided macros, or even the template classes, as long as it can provide a function to create instances derived from the base metamodel interfaces. It would be possible for some component to provide its own implementation of these interfaces.

The metamodel takes care of providing one uniform generic interface, decoupling the interfaces of components from the needs of the various services that compose the framework (Figure 2).

This generic interface can be used not only to discriminate components by their type but also to obtain information about the attributes a component uses, its “needs” or “receptacles” in [4] and [6] respectively. It also provides information about the interfaces a given component implements, thus the “abilities” or “interfaces”,

Table 2: ARToolkit tracking metamodel.

```
EXPORT_CLASS (ARtrack)
ATTR(config_file, String,
                  get ,set, String)
ATTR(fitting_mode, int, get, set, int)
ATTR(imageproc_mode, int, get,set, int)
METHOD1(void,false,
              add_marker,ARmarker*)
METHOD1(int, true,
              load_pattern, String)
METHOD(void, false, detect_markers)
METHOD(void, false, select_optimal)
METHOD1(ARmarker*, true,
                get_marker, String)
METHOD(void, false, resolve_transform)
END_EXPORT(ARtrack)
```

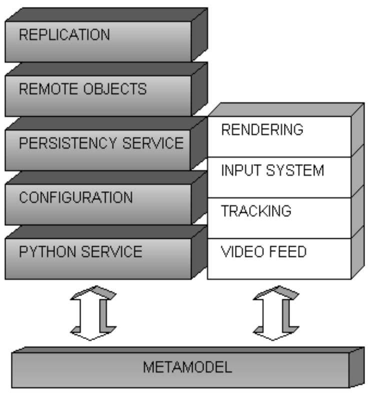


Figure 2: Metamodel decoupling application components from architecture services. The framework services are shown darkened, while the white blocks symbolize example components from the demo in Section 5

also in above mentioned work. Furthermore, the IMethod interface supports dynamic invocation of arbitrary methods. A method instance can store references to arguments, and be executed when necessary, thus providing a way to defer the execution of a call from its preparation generically. It can be linked to different instances of the class. A creation procedure is similar to a method, but the result of its execution is the creation of a new instance of the class (it is a constructor). So each IClass (in the metamodel) is like a factory for its instances. Instantiating the factory pattern [9] is then reduced to storing a collection of ICreation instances. Attributes can also be set or their value obtained by using the appropriate instance of the IAttribute interface.

4. Generic Services

The interfaces defined in the previous section are sufficient to handle a component in a generic fashion. Through these interfaces a number of support services can be created that can be used by all components, even components that have not yet been created. The following subsections include examples of services already available with different levels of maturity in the proposed architecture.

4.1. Communication

As stated at the beginning of the related work section, a distributed system must provide some form of transparency. For this and for flexibility reasons first remote procedure call, and later remote objects have been considered essential for distributed systems. They allow a developer to use an instance of an object (or a procedure) without knowing where it is being executed.

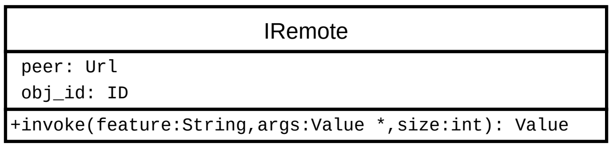
Replication was introduced in the area of fault tolerant systems, but data replication has also proven useful for performance reasons. A number of approaches to distribution in AR [12] and VR [16] [21] have been based on replication of the scenegraph.

CORBA compliant middleware provides access to these services. In the following subsections the approaches taken by this architecture to the mentioned technologies will be outlined. Finally a possible interface to a CORBA infrastructure is introduced.

4.1.1. Remote Objects

Remote object architectures take advantage of the separation between an interface and its implementation to place the interface in one machine, and the implementation in another. A client makes method calls on a *proxy*, which only *marshals* invocations into messages and *unmarshals* reply messages into result values. The implementation resides at a server machine, and provides the same interface. An incoming invocation is first passed to a *server stub* or *skeleton* which *unmarshals* it, invokes the method, and returns the result. Client *proxies* and server *stubs* must be generated for every interface that is expected to work remotely, which is an obstacle if an component that was not designed for remote usage is suddenly expected to move to a remote computer and work.

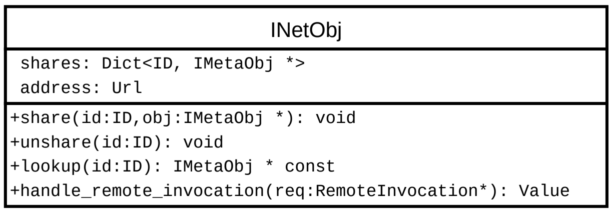
The proposed architecture includes an interface that can act as proxy of any component that supports the metamodel (Figure 3). This *generic proxy* creates a *remote invocation* instance with the given arguments when its *invoke* method is called, it serializes it, and passes it to the underlying infrastructure to be sent by whatever communication method is being used.



```
IRemote
peer: Url
obj_id: ID
+invoke(feature:String,args:Value *,size:int): Value
```

Figure 3: *Generic proxy* interface.

On the server side *skeletons* are not needed. A *generic server* (Figure 4) keeps a list of id of the *shared* components. Upon reception of an invocation, it *unmarshals* it into a remote invocation instance; then it retrieves the indicated component and from it the required IMethod interface; which is subsequently invoked, finally, the result is sent back to the client (Figure 5). At runtime, an object can be *shared* just by adding it to the *generic server* with the desired id.



```
INetObj
shares: Dict<ID, IMetaObj *>
address: Url
+share(id:ID,obj:IMetaObj *): void
+unshare(id:ID): void
+lookup(id:ID): IMetaObj * const
+handle_remote_invocation(req:RemoteInvocation*): Value
```

Figure 4: *Generic server* interface.

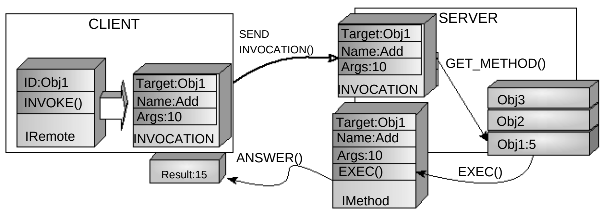


Figure 5: The remote invocation process as implemented by the proposed architecture.

This approach has the disadvantage that the *generic proxy* can not be used in a place where a local component is used (because they might not share the same interface) however it has the advantage that a minimum implementation is used, this could prove useful when developing for resource restricted devices like PDAs or mobile phones. On the other hand, if what is called *static invocation model* is required, wrappers can be generated by simply inheriting both from the exported class, and the *generic proxy* and redefining the methods of the exported class to call the *invoke* method of the *generic proxy* instead. These wrappers could also be generated automatically. A similar approach for Java was presented by [10].

4.1.2. Replication

The discrimination between *passive* and *active* replication gives two replication models. In active replication all *replicas* execute the same operations; while in the passive model, one replica executes the operations and then actualizes all others, during a *synchronization* step. Some scenegraph replication approaches [12] [16] [21] have been accomplished using passive replication.

One of the problems in replication is keeping replicas consistent, of the above mentioned approaches some utilize the *virtual synchronous* model [5], in which replicas are collected by dynamic groups that allow them all to observe the same communication in the same order. In this model, when a new replica *joins* the group the complete *state* must be transmitted to it.

The current architecture also bases its replication facilities in the *virtual synchronous* model and uses a group communication toolkit for reliable multicast. An active replication facility is provided by a single interface (Figure 6), which derives from *generic proxy* and a single *generic server* (which manages only one object, and does not have a share list). Method invocations are sent through the network, while attribute invocations are handled locally. Because the communications are received by all replicas in order, it is guaranteed that executing a remote invocation will keep the replicas consistent.

On the other hand passive replication is supported by providing a wrapper object (Figure 7), the wrapper executes all methods on its current object, and synchronizes the replicas when its method *sync* is executed. A variation, as implemented for the distributed scenegraph is to save incremental change information for each method execution and then send only the changed parts to the replicas. This variation is dependent on the replicated object, and thus cannot be implemented generically.

An object replicated through passive replication requires a synchronization mechanism, otherwise changes could take place at two different replicas, and the synchronization step would leave the replicated object in an inconsistent state. A distributed locking approach could be implemented using the described mechanism of active replication, to guarantee synchronized access to the replicas.

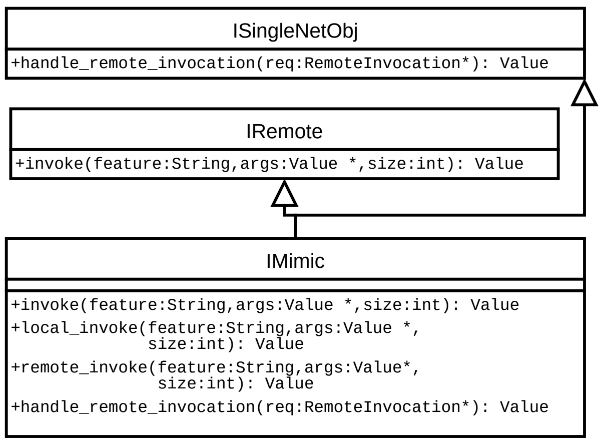

ISingleNetObj
+handle_remote_invocation(req:RemoteInvocation*): Value
IRemote
+invoke(feature:String,args:Value *,size:int): Value
IMimic
+invoke(feature:String,args:Value *,size:int): Value
+local_invoke(feature:String,args:Value *, size:int): Value
+remote_invoke(feature:String,args:Value*, size:int): Value
+handle_remote_invocation(req:RemoteInvocation*): Value

Figure 6: Active replication interface.

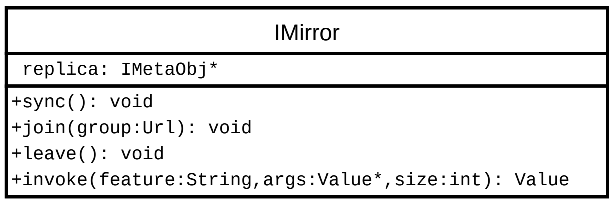


Figure 7: Passive replication interface.

4.1.3. Interfacing with CORBA

As stated above, our approach does not require the generation of *proxies* or *server stubs* for the remote invocation facilities to work. It would be possible though, to generate a CORBA interface, keeping the flexibility of our architecture. A reason for this could be that there are available ORBs which have proven scalable, and provide support for wide range of protocols. Our approach is aimed at simplicity, and as such it may fail to support certain protocols. Neither has it been deployed widely enough to prove its scalability.

To create a generic CORBA interface that would not require to generate *proxies* and *server stubs* for all interfaces, it would suffice to provide an interface for the above described *generic proxy*. The client side would still only have an *invoke* method, as also the server side. On the client side, the invoke method would marshal the arguments as before, but send them through CORBA invocation. On the server side, a server stub would call the *handle–invocation* method of what was described above as *generic server*.

This approach would also provide a minimum implementation for remote objects, this time using CORBA middleware. It has similar advantages and disadvantages with the above. However it has the advantage of using proven communication architecture. This also proves the flexibility of our proposal, as it can adapt to using different models with a minimum impact on the already implemented components.

4.2. Scripting: Interfacing with Python

In the following the *interpreted* language will also be referred as *external*, and code in the interpreter as *external code*, while the *compiled* code will be referred as *local*, and its language as *local language*, without loss of generality.

The expected functionality of a scripting interface includes the following four goals:

1. handling components from the interpreter.

2. creating new components in the interpreter based on the exported ones.

3. running scripts from the local code.

4. handling components created in the interpreter from the local code.

The first two are generally referred to as *extensions*, while the last two are examples of *embedding*.

Interfacing with other languages is done using some kind of *wrapper*, which serves as glue between the local and external languages. Wrappers must be registered with the external language. These wrappers are responsible for receiving invocations from the external language, translating these invocations and executing them on the local language, and of returning appropriately to the external language. This applies to most interpreted languages, however, in the following, the discussion will be restricted to object oriented ones.

The focus is then what is needed for the wrapper to translate between the two languages. There are different approaches to creating wrappers for external languages, (the following will be focused in Python). In general the languages provide an API which can be used to write extensions [22]. A wrapper for a simple class (Table 3) using the API would look like Table 4. There are also libraries that provide automatic *static* translation, and can generate glue code, although they are not completely unobtrusive (that means they require developers to write in a certain way)(see Table 5). However, the major disadvantage is that these libraries can only make a *static* translation, that is if changes are made to the exported interfaces, code must be regenerated and recompiled. This defeats the flexibility goal, since changes in a component now impose changes in the service, which no longer is *generic*. The idea of replacing an interpreter with a different is also unimaginable without generating code for every possible interpreter, just in case.

Table 3: A simple class that will be exported to Python.

```
class World{
 public:
  void set(std::string m){ msg=m;};
  std::string greet(){return msg;};
  std::string msg;
  };
```

In the proposed architecture, to achieve the first goal a *generic wrapper* Table 6, based on the metamodel interface has been created. It receives invocations from the external language, with arguments also in the external language, queries the metamodel for the requested feature and either executes it with the given parameters if it is a method, or manipulates its values if it is an attribute. When returning objects which are themselves components, they must also be wrapped to pass them to the interpreter. The wrapper accomplishes the task of translator between the framework and the external language.

A *generic Python service* (Figure 8) manages the dynamic wrapping of components, the creation of instances, and their destruction,

The interpreter also requires to know about the exported *classes* to be able to accomplish the second goal above. In the case of Python, the interpreter relies on reflection, by which all its classes are defined. Taking this into account, the *generic Python service* generates Python classes from the framework classes at *runtime*. This service also creates wrappers, as only it knows about the counterparts of the local metamodel in the interpreted side.

Table 4: Wrapper to export World class written using the API, it doesn't include the declaration of methods other than constructor and destructor. Neither does it show the declaration of the PyWorldType structure, for reasons of space.

```
typedef struct {
    PyObject_HEAD
    World *world;
} PyWorld;

/*--------- Destructor ------------------------*/
static void
PyWorld_dealloc(PyWorld* self){
   delete self->world;   }
/*--------- Constructor ------------------------*/
static PyObject *
PyWorld_new(PyTypeObject *type, PyObject *args,
                              PyObject *kwds) {
   PyObject * result = PyObject_New(PyWorld, type);
   result->world = new World;
   return result; }
```

Table 5: Wrapper written to export World class with Boost.

```
#include<boost/Python.hpp>
using namespace boost::Python;
BOOST_PYTHON_MODULE(hello)
{
   class_<World>(''World'')
      .def(''greet'', &World::greet)
      .def(''set'', &World::set);
}
```

Table 6: Generic Wrapper declaration.

```
typedef struct {
    PyObject_HEAD
    /* reference to its Python class*/
    PyMetaClass   *in_class;
    CPtr<IMetaObj> obj;
} pyObj;
```

Regarding the third goal, the *generic Python service* runs scripts on behalf of the framework, and can decide whether the interpreter, or the framework should be initialized; this last is crucial, as the framework interface could be loaded from the interpreter by including the shared library, in this case, the first object created is the *generic Python service* and it must be aware of the need to initialize the basic infrastructure before running any commands.

The last goal is to be able to invoke methods on instances created by the external language from the compiled one. This is again possible through a proxy, and it is only possible to do it directly if the interface being invoked has been defined in the local implementation. For example if the instance, created in Python, is derived from an interface defined in C++, then it may be manipulated through this interface, if not it might only be manipulated through a proxy.

4.3. Configuration

Configuration is possible by utilizing the interpreter. Through it, a component can be queried for its interfaces, and its attributes. Based on this information, and the information kept by various subsystems on the runtime components, it is possible to assign a different instance of the same interface to a given component.

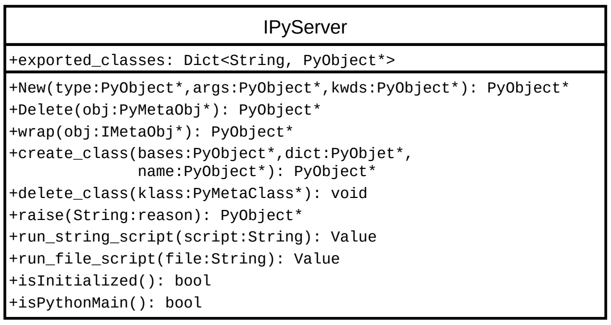

```
IPyServer
+exported_classes: Dict<String, PyObject*>
+New(type:PyObject*,args:PyObject*,kwds:PyObject*): PyObject*
+Delete(obj:PyMetaObj*): PyObject*
+wrap(obj:IMetaObj*): PyObject*
+create_class(bases:PyObject*,dict:PyObjet*,
            name:PyObject*): PyObject*
+delete_class(klass:PyMetaClass*): void
+raise(String:reason): PyObject*
+run_string_script(script:String): Value
+run_file_script(file:String): Value
+isInitialized(): bool
+isPythonMain(): bool
```

Figure 8: *Generic Python service* interface.

There is a local naming service which provides directory-like structure and manipulation for searching components by name.

This interface is very primitive; therefore a GUI based interface with semi-automatic assistance is planned for future work.

4.4. Serialization

One common usage for introspection has been serialization, the capability to save an object state to a stream from which it can later be recovered, be it at the same location, or at a remote one. Our architecture also provides a serialization interface which works automatically for most components. However, if a developer is in doubt, she may decide to implement the methods *save* and *restore*, which will be used in such a case. These methods are passed an instance of a *Store*, which provides methods to store and restore basic elements such as *int, float, bool,* etc. and facilitates the task of saving/restoring current state.

5. Examples: Reconfigurable city planning

An example application has been created, which allows two users to select, place and manipulate objects in the frame of a city-planning application. For this demo, extra infrastructure was required for distributed presentation. A distributed scenegraph was created to fulfill this requirement. The distributed scenegraph approach is briefly introduced in the next subsection, followed by the application layout.

5.1. Distributed scenegraph

As a test of the communication facilities, and with the goal of providing a distributed presentation facility in the framework, Coin3D (Open Inventor implementation) was extended with *distributed node*. The architecture of Open Inventor allows receiving notification on modification of a subtree, by placing its root in a sensor node, based on the approach taken by [12]. The *distributed node* places itself into a sensor to receive notification of the changes in its subtree. Furthermore, it uses the *passive replication* facilities to relay the changes to remote replicas.

There are different ways to relay the changes. It can be done upon occurrence automatically, or by explicitly calling a *synchronize* method on the node. The second approach is desirable when changes to the subtree occur in bursts, then the changes can be kept in a list, and sent all together. This reduces the transmission of small messages. Nevertheless, the delay produced in the changes to the replicas must not be noticeable to the viewers.

Among the advantages of this approach at distributed scengraph are the possibility to compose a scenegraph from more than one remote scenegraphs (as this approach provides *partial sharing*) and the usage of a well-documented scene-toolkit. However, the proposed approach lacks concurrency management facilities, which must be added separately. A shortcoming of using an external toolkit, which does not support the proposed architecture, is that it can not benefit from the rest of the services (i.e. scripting). Solutions to the last issue are under analysis.

5.2. Application layout

Users sit in front of a table, the playground table, which contains markers and cameras pointing to its surface, one camera for each host is used. A special marker with a handle is used as an input device (the stick-marker, Figure 9). Interaction is gesture based, dependent on the motion of the stick-marker. Movements that trigger actions include inclination of the marker by more than a given angle, up movement of the marker and others. The markers are tracked by ARToolkit. Users may change the position of the menu, by moving the marker that symbolizes it, open it (by occluding it with the stick-marker) and select models from a comprehensive directory menu (Figure 10). Once a model is on the stick-marker, it may be dropped at any point on the table surface.

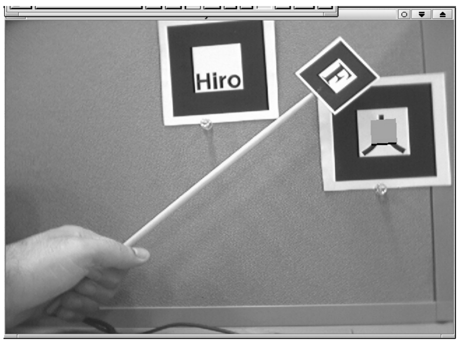


Figure 9: The Stick-Marker input device

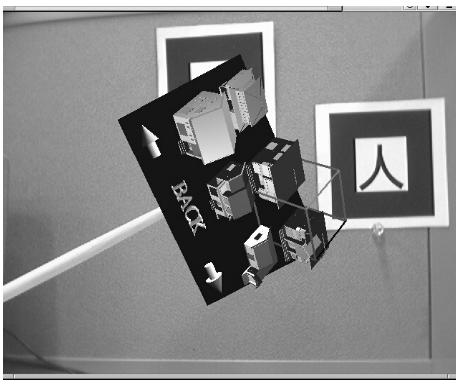


Figure 10: Model selection menu on stick-marker

Configuration changes can be scheduled at any time by issuing commands in a Python shell-like console.

The demo is pointed not really at using the application, but at seeing how it can be reconfigured on the fly. This is only an example of possible uses for the described architecture. And although a similar demo could be created by using other approaches; creation of software for a seemingly complex setup as this is greatly simplified by the proposed architecture, as the software components must focus on their own task, and are required to solely support the metamodel interface.

There are several ways one could divide the application. Taking into account layers relevant to AR systems, it can be divided vertically and horizontally as in Figure 11. The demo uses three computers, two clients, and a server that manages the shared resources, (shared scene).

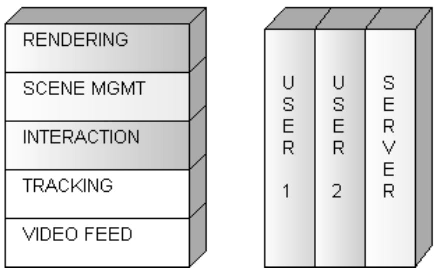


Figure 11: Layering of the demo application.

With the exception of the shared scene management, the other layers could be handled by any of the computers, client or server, that is depending on whether they have access to the underlying layer. For example: *tracking* depends on the *video feed* to recognize the markers in the current frame. If a host has access to *video feed*, either local or through the network, it can handle tracking. Horizontally, any host capable of doing its own *tracking* can handle interaction for a user, and thus be separated from the server.

There are then a number of deployment scenarios for the demo, starting from a standalone server, handling everything (Figure 12); to a distributed system with one host per user, and the server handling only parts common to all users (Figure 13).

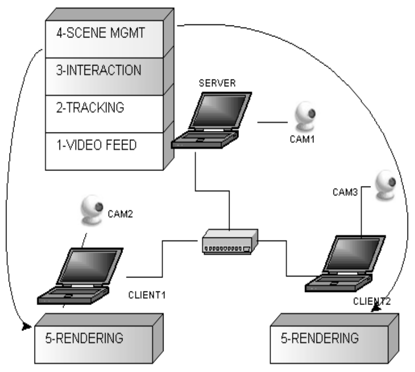


Figure 12: Standalone server handling everything, with two rendering slaves.

One of the ideas behind this, is that users without access to video, because they lack the hardware or software, can use the application by connecting to one of the hosts that can provide it. Another idea is that future capture devices with higher processor power could probably capture the image and send it through network to one server. By using this approach, a server can be connected to multiple cameras.

These are only some of the possibilities considering the above division for the application. Some ideas that are not proven in this demo include how changing some parts of the infrastructure does not affect the implementation of the components that are used by it. For example, it would be possible to replace the Python service with another interpreted service, (Tcl, Ruby, etc), and the rest of the components would not need to be changed. It would even be possible to have a different interpreter running in each client, this has not been included in the demo for lack of implementations of services to handle other interpreters.

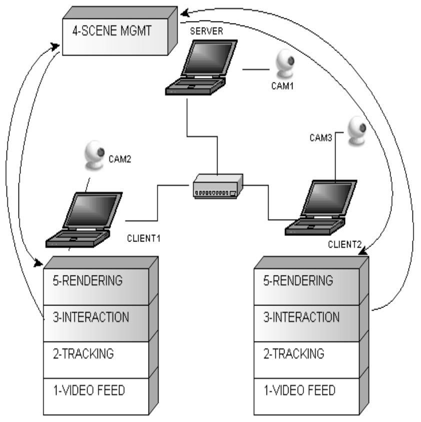


Figure 13: Separate systems for each user leave the server to handle only shared parts.

6. Conclusions and future work

An architecture providing different strategies to handle the challenges described by [14] has been implemented. Our own goal of making this an unobtrusive framework has proven to be against some of the requirements for adaptability, and flexibility. Realizing these would be a tradeoff, we settled to create an interface general enough to decouple the most obtrusive requirements from the implementation of the framework, obtaining what we believe is possibly the less obtrusive interface in the present metamodel.

On the way to the development of this architecture, we have proven the usability of a metamodel for *dynamic generic programming*, and given examples in the creation of a set of *generic services*. These are not the only possible *generic services* that can be created through the metamodel interface, but they are the beginning of a growing set of services that do not need to adapt, because they are already *generic*.

The design of software architectures in different areas has followed the idea of providing a frame, based in known practices in the area, to delimit future applications, allowing faster development by reusing well-known-solutions. Following this approach, usually meant making a great number of design decisions, based on the well-known-practices of the area, that would later lead to the frame specified previously. The proposed solution departs this approach radically by providing an extensible base upon which particular solutions can be built. At first sight, it may seem that this approach does not provide a real solution, as many design decisions have deliberately not been taken. On closer inspection however, the advantage of such infrastructure starts to be clear, as decisions can be delayed and solutions tailored for particular situations at runtime by plugging in components with the desired behavior.

At the time of this writing, porting to OSs other than Linux (Windows and WinCE) had already been started. However, even when the described interfaces (metamodel, remote objects, replication, serialization etc.) have already been ported, the rest, mostly OS specific part is currently unfinished.

The configuration service is still an evolving part, and it doesn't actually include the notion of different contexts; a topic to be tackled in future work.

The last issues are the remaining obstacles before our work can be tested in mobile applications.

Acknowledgments

This study was supported in part by the Core Research for Evolutional Science and Technology (CREST) Program - ”Advanced Media Technology for Everyday Living” - of the Japan Science and Technology Corporation (JST).

References

[1] Java tm core reflection api and specification. Technical report, Sun Microsystems Inc, 1997.

[2] ”java remotemethodinvocation specification”. Technical report, Sun Microsystems Inc, 2002.

[3] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. ”volume ii: Technical concepts of components-based software engineering”. Technical report, Software Engineering Institute, Pittsburgh, PA 15213, USA, 2000.

[4] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stefan Riss, Christian Sandor, and Martin Wagner. Design of a component-based augmented reality framework. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, October 2001.

[5] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *SOSP ’85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 79–86, New York, NY, USA, 1985. ACM Press.

[6] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 160–178, London, UK, 2001. Springer-Verlag.

[7] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Gian Pietro Picco, and Stefanos Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC’05)*, Berlin (Germany), September 2005. To appear.

[8] G. Blair F. Kon, F. Costa and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[10] Ennio Grasso. Jrb: A reflective orb. Technical Report 14, CSELT, 1997.

[11] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Gerd Hesina, Dieter Schmalstieg, Anton Furhmann, and Werner Purgathofer. Distributed open inventor: a practical approach to distributed 3d graphics. In *VRST ’99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 74–81, New York, NY, USA, 1999. ACM Press.

[13] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[14] Asa MacWilliams. Software development challenges for ubiquitous augmented reality. In *GI Augmented Reality and Virtual Reality Workshop*, September 2004.

[15] Bertrand Meyer. *Object-Oriented Software Construction*. Computer Science Series. Prentice Hall, 2 edition, 1997.

[16] Martin Naef, Edouard Lamboray, Oliver Staadt, and Markus Gross. The blue-c distributed scene graph. In *VR ’03: Proceedings of the IEEE Virtual Reality 2003*, page 275. IEEE Computer Society, 2003.

[17] Wayne Piekarski and Bruce H. Thomas. An object-oriented software architecture for 3d mixed reality applications. In *ISMAR ’03: Proceedings of the The 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 247–256, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Manuel Roman, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. Technical report, Champaign, IL, USA, 2000.

[19] Dieter Schmalstieg, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavári, L. Miguel Encarnação, Michael Gervautz, and Werner Purgathofer. The studierstube augmented reality project. *Presence: Teleoperators & Virtual Environments*, 11(1):33–54, 2002.

[20] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[21] Henrik Tramberend. Avocado: A distributed virtual reality framework. In *VR ’99: Proceedings of the IEEE Virtual Reality*, page 14. IEEE Computer Society, 1999.

[22] Guido van Rossum. Extending and embedding the Python interpreter. Technical report, 2005. For Python Release 2.4.1.

[23] K. Watsen and M. Zyda. Bamboo - a portable system for dynamically extensible, real-time, networked, virtual environments. In *VRAIS ’98: Proceedings of the Virtual Reality Annual International Symposium*, page 252, Washington, DC, USA, 1998. IEEE Computer Society.

[24] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.