D5c Client Platform Implementation and Behavior Definition Tools

Date:	15-August-01
State:	Draftl
Work Package:	WP5
Participant Partner(s):	NR
Author(s):	Per Thomas Jahr (NR), Joachim Lous (NR), Shahrzade Mazaher (NR), Anders Moen (NR).

1 Introduction

This document describes the design and implementation of the MADISON simulation framework prototype. This prototype consists of three parts:

- A client side simulation framework developed at NR,
- A server side framework implemented by Archetypon, and
- A network module developed by Boostworks.

This document is mainly concerned with the client simulation framework and will not elaborate on the other parts of the prototype. For a detailed description of the server framework, the reader is referred to [5].

The client simulation framework consists of an object model and an infrastructure that is common to all simulation applications. It is closely related to the simulation object's behavior, and must be in place in order to support any behavior definition toolset.

The client simulation framework is concerned with the object model for the simulation applications, a model of interaction between the distributed simulation objects, services required by all application (e.g., an object registry, etc.), and specific common functionality required by the simulation objects of a distributed, multi-user simulation, such as dead reckoning and predictive algorithms.

This document emphasizes on the changes in the client side simulation framework compared to the version described in an earlier deliverable [4].

The terminology used in this document is the same as explained in [4]. The rest of this document is organized as follows. Section 2 sketches briefly the MADISON architecture. Section 3 describes the simulation model adopted in MADISON. Section 4 deals with the design of the MADISON framework while Section 5 is concerned with its implementation. Section 6 describes some rudimentary authoring tools developed by the project that can form the basis of a more advanced toolset.

2 MADISON Architecture

Figure 2.1 depicts the general architecture of the MADISON system. For details on the architecture, the reader is referred to [3]. The simulation platform at the client interfaces on the one hand to the network to receive events sent from other clients and on the other hand to the MPEG-4 module of the set-top box via the MPEG-J/EAI interface. This MPEG-J interface is a Java interface defined by the MPEG-4 standard [1] to enable interaction between applications and the VRML scene graph while the EAI (External Authoring Interface) [2] is a corresponding interface defined by the VRML standard. These interfaces are quite similar in functionality and the choice is mainly based on which one is supported.

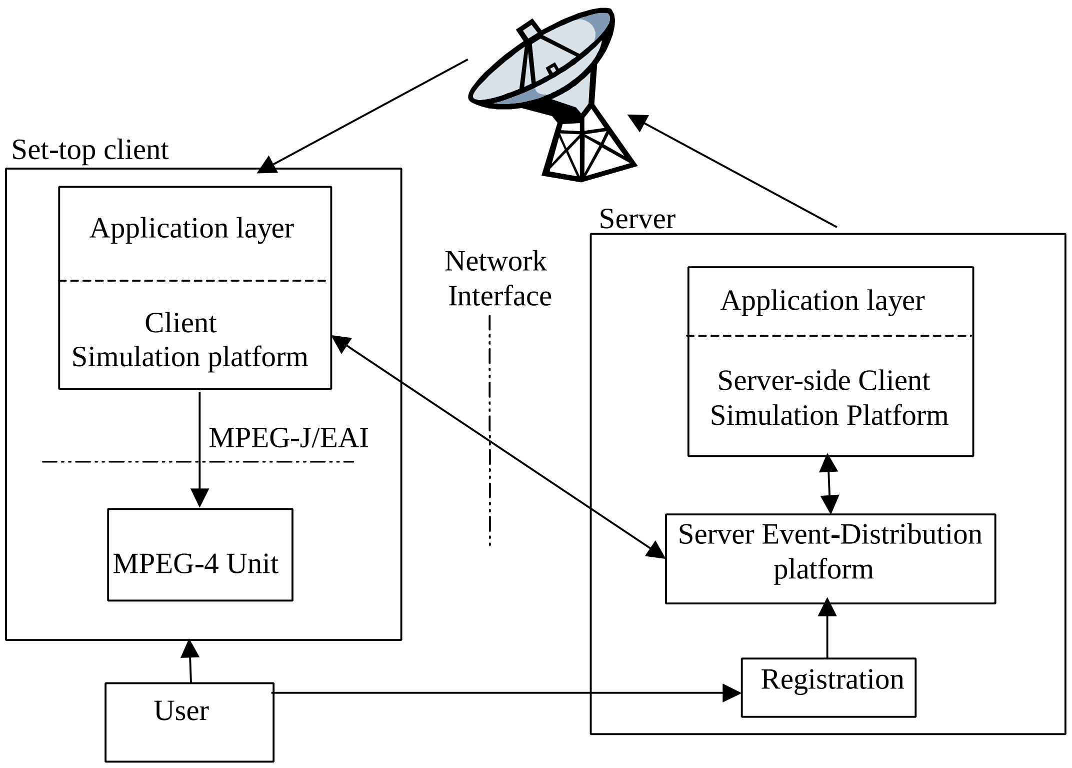


Figure 2.1 General Architecture of the MADISON System

Note that the broadcast communication is between the content server, which can be the same as the simulation server, and the MPEG-4 unit of the set-top box.

3 Simulation Model

This section gives a short description of the MADISON simulation model, described in more detail in deliverable [4]. The main features of this model are given below.

3.1 Distribution Model

The distribution model chosen in MADISON is a client-server architecture where all clients are connected to a central server and all communications between the clients goes via this route, i.e., there is no direct peer-to-peer communication between the clients.

The simulation itself is not centralised, in the sense that the server does not hold the complete *true* simulation state; this is distributed among the clients. The simulation part of the server is a regular client with some special objects. But in addition to this aspect, the server has a message distribution module whose only purpose is to filter and forward messages between the clients.

3.1.1 Replication, Pilots and Drones

There are two types of simulation objects (*actors*) populating a simulation: avatars and bots. Avatars represent users, bots are controlled programmatically. Both actor types can appear in two roles: An actor has a *pilot* replica on exactly one client, which makes all final decisions about its behavior, and zero to many *drone* replicas at other clients where it can be observed. To make the drones replicate actions and state changes, pilots use platform services to send them special update messages called PDUs.

Note that there can exist bots with only local scope. That is, they have no drones and no direct effect on the global state of the simulation; they only matter on the client where their pilot resides. Typical tasks are local application administrative logic, or tools whose state is only relevant to each client, such as navigation aids etc. displayed in the HUD of the user.

The server contains the pilot replicas of normal bot objects if any, drone replicas of the avatars, and of special "system" bot objects, e.g., a scorekeeper or other global administrative logic. Each client contains the pilot replica(s) of the corresponding avatar(s), drone replicas of the avatars of the other clients, drone replicas of the global bot objects, and the only replicas of each of the local bot objects.

We thus have these types of actor objects at a MADISON client on a user machine:

- a pilot replica of the avatar controlled by the user,
- drone replicas of (a subset of) avatars controlled by other users,
- drone replicas of (a subset of) bot objects, and
- (the only replica of) bot objects with local scope.

The set of objects on the server-side client comprises of the following kinds:

- a pilot replica of each of the bots (with global scope) existing in the simulation, and

- drone replicas of all avatars controlled by the users.

3.1.2 Zones

The mechanism for the different actors to specify their area(s) of interest is called *zone*. A zone has a set of member actors, and the messaging server only forwards messages addressed to a zone to clients with pilots or drones that are members of it. In MADISON, a zone is an abstract notion; the semantics of being a member are wholly application-specific. Although the conventional geographical area type zone is the most common use, zones can also represent more abstract notions, such as affiliation to defined group of avatars or access to a particular type of events not dependent on spatial position.

Each actor has a *home-zone*, the zone corresponding to its geographical location in the simulation world, of which it is automatically a member. Actors can become members of other zones of interest by subscribing to them. All the messages an actor generates are stamped with its home-zone, and are distributed to that zone's membership. Moreover, an actor can specify additional abstract zones for a given message to be distributed to.

This mechanism allows many to many relationships between actors and the zones. That is, an actor can send events to many zones and also can receive events from many zones.

3.2 Interaction Model

In MADISON, from a logical point of view, all types of communication between the different simulation objects (actors) are *local*, carried out in parallel on each client machine where the actors are represented. That is, when a pilot actor initiates a communication, the (inter)action is performed locally in the "eyes" of actors on the same client machine. Separately from this, the pilot actor will also often use the dispatcher services to inform all its drones at other clients of the communication, but this is a "private" communication internal to the pilot implementation. The distribution aspect cannot be observed directly by other actors inside the local simulation arena; they only "see" each other as they are represented locally. When receiving the order, each of the remote drones initiates the same (inter)action locally inside *its* local environment, as if it was acting autonomously there as far as the other actors on the remote client are concerned.

There are two mechanisms for local interactions, described in the following sections.

3.2.1 Exposed Fields

Actors may let any of their state values be publicly readable, so that other actors can programmatically find out things about them, or monitor what they are doing. A typical property that almost all actors are required to publish is their position, but others may publish anything they like. The set of exposed fields is identical for pilot and drone instances of the same actor class (although the values may be out of sync); the surrounding actors reading the fields should not need to know or care which role it is in.

3.2.2 Actions

Actions are a more concrete form of local interaction between actors. Actions are initiated by the pilot replicas of actors, either based on their internal logic in the case of bot actors, or under the control of the end-users in the case of avatar actors. There are two subtypes: Independent actions involve nobody else; examples of independent actions are waving, sitting down, etc. Directed actions are also initiated by the pilot replicas of actors but are directed towards other actors; examples are moving an object, pushing a button, etc.

Independent Actions

The other actors at a client are informed of an independent action by a remote pilot only when the local drone replica of that actor carries out the action, having received the corresponding information from its pilot. For avatars, this information is usually just left for the user to observe through the normal presentation of the acting drone.

The situation is different for pilot bots. They do not have a controlling end-user who could observe the independent actions carried out by other actors and, if necessary, react to them. They must therefore be notified programmatically when independent actions happen, either by having registered to listen for such actions (not implemented yet), or by polling exposed fields of the local drone for changes resulting from the action.

Directed Actions

A directed action has a *subject* (the pilot actor that initiates it) and an *object,* or *target,* (the actor on which the action is performed). Sometimes the action may also refer to *secondary objects*, i.e., other actors used or involved in the task, but those are just considered like any other parameter of the action.

Usually, the target actor is a drone. The target drone may carry out its reactions to the action, using prediction if necessary, but that is only a local and temporary change. The subject pilot also informs its drones to carry out the same directed action on the same target actor, and one of those drones will end up performing the action on the actual pilot of the target actor. The target pilot will react correspondingly, and only when it transmits its reaction to its drones will they know the 'official' outcome. The drones of the target actor must then locally reconcile their predicted reaction with the update from the pilot if necessary, and the communication loop is complete.

4 Actor Implementation

This section is concerned with the internal implementation of actors in the MADISON framework. It first describes briefly the parts that have been elaborated on in [4], and then details the new components of the design.

4.1 The Actor Object Structure

The actual classes containing the simulation code can be roughly divided into two groups: the active simulation objects (Actors) themselves, with all their superclasses and owned objects, and the 'platform' including the standard system services provided to support the actors. This division corresponds roughly (although not entirely) to the division between static system software and application specific code.

All actors represented on a client appear as object instances descended from the *BasicActor* superclass. Every actor type has a distinct actor subclass to represent it. This class defines the simulation API of the actor type, including the exposed fields it has, the actions that can be performed on it, etc. The rest of the functionality (logic, animation, using services) is delegated down to specialized objects owned by the actor.

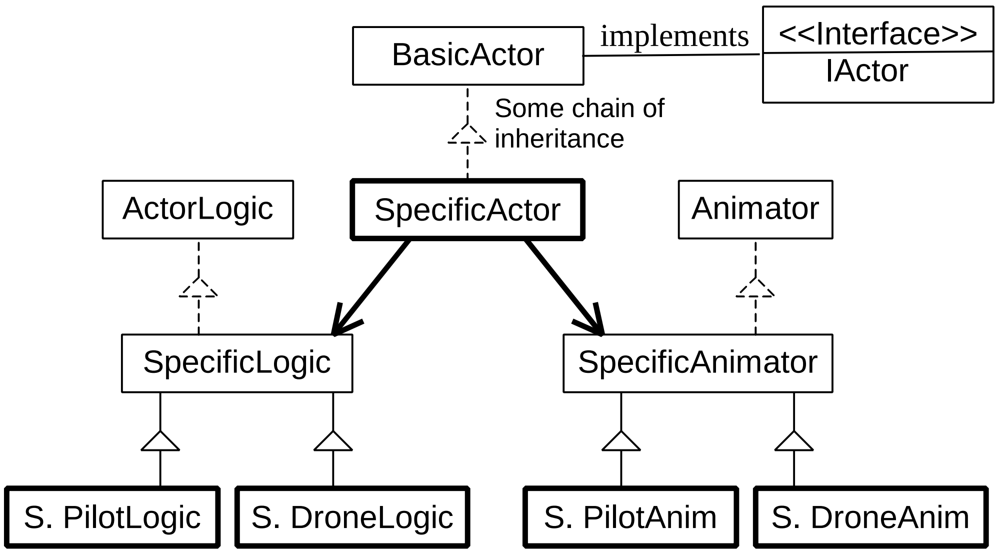


Figure 4.1 The Actor Object Model

Pilots and drones of the same actor use the *same* top-level class to represent them, so that the world outside the object (the other actors in the local simulation) usually does not care which version it is working with; they look just the same to their environment. The only difference lies in which versions of the owned objects are instantiated and used, i.e., either pilot or drone versions. The various versions of the owned objects provide the same interface to the main actor object, but their internal workings can vary widely.

4.1.1 Logic

The logic object is in charge of any change or event at the simulation level. A logic object implements either the simulation role of a pilot or that of a drone. It completely encapsulates the logic, including the distribution aspects, specific to each actor.

Typically, a pilot logic object will be concerned with taking decisions internally (either according to rules/AI, or by getting input from a user or some other external entity), and will synchronize with its drones by notifying them of its state and decision, using the system services.

A drone logic object is mainly concerned with effecting the orders received from its pilot (state changes and external actions), displaying predicted behavior in between the updates, and reconciling the prediction with the true state when updates arrive.

4.1.2 Animation

The animator object handles everything to do with the visual aspect (display) of the actor object, and is the only part of the system that has any knowledge of the player and the scene graph. The normal external view of an object as seen by all other participants is handled by the drone animator, according to the higher-level instructions received from the logic object. In between receiving instructions from the drone logic, the drone animator uses predictive methods to achieve a smooth visualization of the drone avatar

The animator of the pilot object is very different. For a bot, it will usually have no animator at all: the code that controls it has no use for a graphical rendering, and everyone interested in seeing it has a drone copy. For user-controlled actors, the pilot animator will be responsible for presenting visual feedback to the user, such as any dashboard or head-up display-like controls, which the external observers of the actor do not see.

4.2 Interaction

All actor objects hold a reference to the local *services* object, from which they can obtain references to the individual services it encapsulates. One of these services, the *registry service* can be used to obtain references to the local copies of the other actors, based on their global ID or some selection criterion (the ID may have been obtained from an incoming event, or through selection of an object visible on the screen).

Once a local reference to another actor is found, the first actor (the *subject*), can inspect the class, and call public methods on the other actor (the *object* actor), corresponding to observing its exposed fields.

Using the local reference, an actor (usually the pilot), can also perform actions on another actor. Each action supported in an application should be defined in a class implementing the interface *IAction,* which is a fixed part of the framework. This interface consists of a set of methods that can be grouped as follows:

1. Presentation methods to be used by the owned *animation* object of the actors to present any default animation of the action. Note that one action can have up to four presentations: subject (pilot), subject (drone), object (pilot) and object (drone).

2. Check methods to check whether the action is legal for the subject/object (based on their combination of types and/or states)
3. Logic update methods to be used by the owned *logic* object of the actors to enact the default logical reaction (state update) involved in this action.

This means different actors don't all have to duplicate similar code for the same common reactions, they can simply inherit a standard series of calls to the respective methods as the their standard reaction to all or most actions. The default implementations are provided as part of the specific action class rather than inherited down the actor hierarchy, so that new actions can be added without modifying any actor classes.

Of course, the logic and animation classes of an actor may choose *not* to use the default implementations, but rather provide their own specialized behavior. For a 'push' action, most stationary actors would fail the test and thus ignore the action. Most others might use the standard reaction, which moves them a short distance in the appropriate direction. Whereas a bot representing a button might skip the test and override the animation and logic, and in stead remain in place, display a custom animation, and open a door. The selection of a target actor for an action is the responsibility of the subject actor.

Figure 4.2 depicts the implementation of the MADISON interaction scheme described in Section 3.2.2 at the distribution level. The subject actor (SP), a pilot, on *Client1* sends an action object of the desired class, represented by the "*Some Action Object*" box in the figure, to the object actor (OD), a drone, on *Client1*, which possibly carries out a predicted reaction to the action. SP also sends a *DirectedAction* PDU, a relay of action event, to its drone (SD) on *Client2*. On *Client2*, the subject's drone recreates the action specified in the received PDU, and sends it to the local replica of the object actor (OP), also specified by the PDU. This replica, being a pilot, carries out its side of the action, and updates all its drones accordingly by sending to them a *StateUpdate* PDU. The drones of the object actor, ODs, must then reconcile their predicted state with the actual state of their pilot.

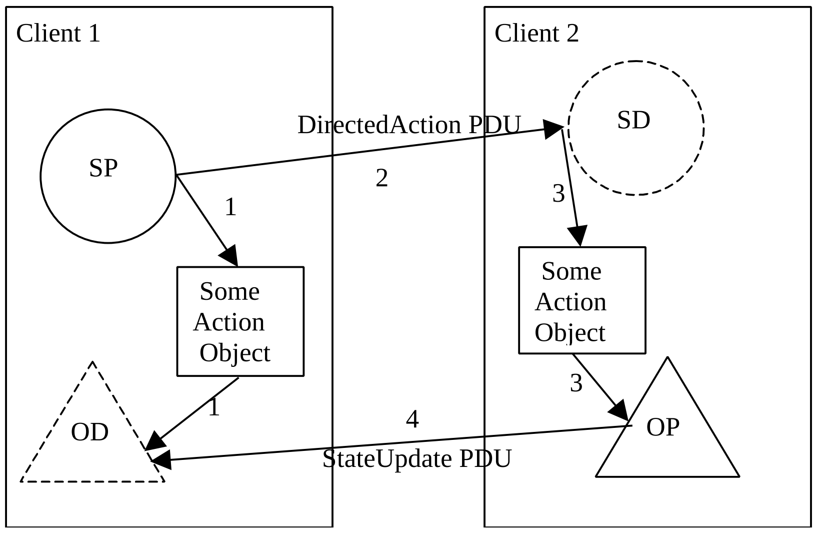


Figure 4.2 Overview of Interaction between Actors

At the local level, on Client1, the SP object, the originator of the action, performs the following steps:

1. obtains the local address of the target actor, OD, through the services offered by the *ActorRegistry* object of the client platform, described in the next section, Client Platform Implementation;
2. instantiates an action object of the desired class;
3. delivers it to the target actor by invoking on it the *processAction* method, defined in the *IActor* interface, with the action object as parameter.

The OD object, the object of the action, performs the following steps:

1. passes down the action object to its owned logic object;
2. the logic object performs the action, based on its current state, either by using the default methods provided by the action object or by overriding them by its own methods. That is, it updates its logical state and if necessary passes the action object to the owned animation object of the corresponding actor;
3. the animation object follow the same procedure as the logical object but with respect to presentation.

On client2, the SD object having received a DirectedAction PDU from its pilot SP, performs the following steps:

1. passes the PDU on to its owned logic object;
2. the logic object inspects the type of the PDU and acts accordingly.;
3. in this case, the PDU being of the DirectedAction type, it obtains the address of the target object, specified in terms of the actor's unique ID in he PDU, using the ActorRegistry object of its client platform;
4. it instantiates an action object of the appropriate class;
5. delivers it to the target actor by invoking on it the *processAction* method, defined in the *IActor* interface, with the action object as parameter.

This time the target OP is a pilot object that processes the action object following the same steps as described above for the OD object. In addition, after having performed the action, it sends a StateUpdate PDU to all of its drones to synchronize. The OD drone on Client1, upon receiving the StateUpdate PDU, must, if necessary, reconcile its state with that of its pilot.

Each action object is tailored to the function of the action and carries relevant information. For example, a *TeleportAction* object has the position of the destination. This position can be passed as a parameter to its constructor when instantiating it.

5 Client Platform Implementation

A lot of the behavior of any object in Madison is at least partly defined by the common static framework of interfaces, conventions and services that we call the *client software platform*. It provides an essential toolset of predictable code patterns and ready-made functionality for external authoring tools to work with. This section provides some details about the platform implementation of the client side. Most of it applies equally to the server-side client, except the startup sequence and user I/O, which are of course not needed there.

5.1 Startup sequence

For any application of the Madison platform, a large portion of the java code will always have to be application specific, such as the logic and animation implementations of the avatars, and the behaviour of any bots, including possible global “game rules” or “referee” objects. These applications are downloaded to the set-top boxes whenever the end-user wants to use them. For purposes of future upgrades and enhancements after the set-top boxes are deployed, it is an advantage that the more static parts, the platform classes, can also be downloaded in the same fashion. Allowing this also opens the door to running entirely different pieces of software if so desired.

5.1.1 Java Embedding

Dynamic adaptation to each application is achieved by transmitting class files as part of the broadcast stream (which also carries the application specific media objects and possibly an event stream), so that the STB can download client code when entering a Madison-enhanced channel.

The download mechanism consists of a custom class loader, equipped to extract and load named java classes from the stream into the java virtual machine. This is used exactly like the standard file- and http-based class loaders, and is transparent to the client code.

MPEG-4 specifies how to embed java code as separate objects in the multiplexed MPEG-4 stream. However, the third-party MPEG-4 systems coder/decoder we are using (and to our knowledge, all others in existence) are still under development, and do not currently support this part of the standard. As a temporary solution therefore, one of the partners in the project, Archetypon, has implemented a functionally equivalent system which codes the java classes as long numeric-array fields in hidden nodes in the scene, which when received on the client are retrieved through EAI and decoded back to loadable classes.

This alternate strategy makes no difference to the rest of the application: loading is in both cases handled transparently by a class loader, and in both cases it takes one iteration of the broadcast carousel to be certain that all the classes have been received. Consequently, replacing the class loader with one that uses the standard mpeg-4 technique when it becomes available should be a small task, and will not affect the rest of the code base.

5.1.2 Bootstrapping

With all this flexibility, there needs to be a fixed point of contact to initiate loading in the first place. Madison has a very simple yet powerful specification for this: The stream *must* contain a class with a specific name (madison.Bootstrap), and single-function interface, which the receiving client uses to start the chain.

In the implementation, this is used in the following manner: When switching channels, resident software in the box detects if the new channel is Madison-enabled. If so, a small pre-installed java program is started. Its only job is to instantiate the custom class loader (also pre-installed), use it to fetch the application-specific Bootstrap class from the stream, and run its startup method. What happens next is in principle entirely up to the implementation of the broadcast Bootstrap class, but for the normal Madison setup, it instantiates the services object (which effectively builds the entire platform infrastructure), and then loads an application specific control object, which instantiates local actors such as the avatar pilot and initiates registering with the server, thus arriving at normal operation. Incoming PDUs after registering with a zone will take care of all other instantiation.

The platform classes can be transmitted as part of the dynamic code, or be pre-installed; this is just a matter of distribution policy and does not affect the code. A more advanced class loader might also be extended to transparently cache downloaded classes on a hard disk if available, but this is not implemented in this prototype.

5.2 Platform Services

The simulation platforms both at the client and the server are shown in the diagram below. The client platform comprises the *Services* module supporting the flow of the events in the simulation and the *other modules* entity grouping a set of services that support both the logical and the visual aspects of an object's behavior, such as dead reckoning, collision detection, etc. Each application has a handle to the Services module and can thereby use the services it offers.

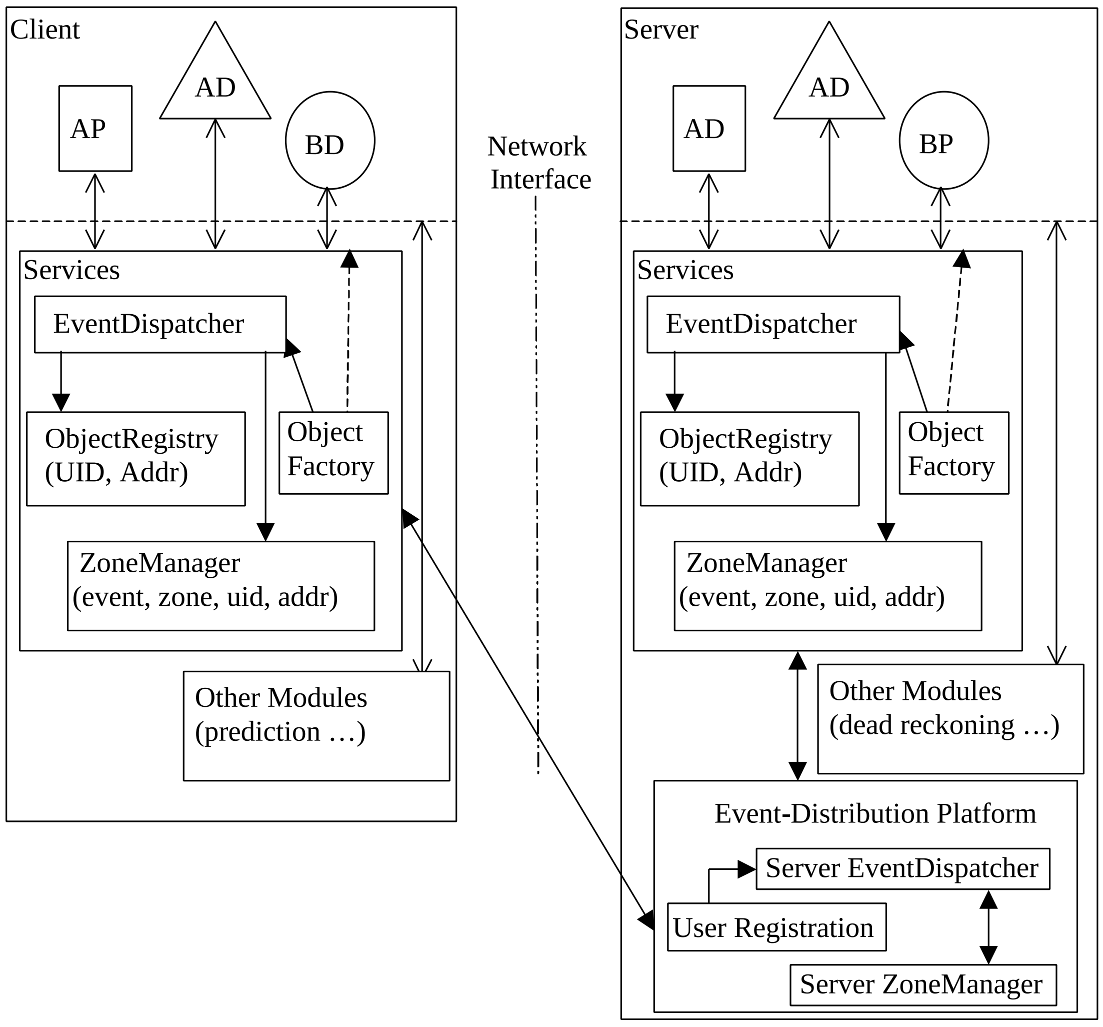

Client
AP
AD
BD
Services
EventDispatcher
ObjectRegistry (UID, Addr)
Object Factory
ZoneManager (event, zone, uid, addr)
Other Modules (prediction …)
Network Interface
Server
AD
AD
BP
Services
EventDispatcher
ObjectRegistry (UID, Addr)
Object Factory
ZoneManager (event, zone, uid, addr)
Other Modules (dead reckoning …)
Event-Distribution Platform
Server EventDispatcher
User Registration
Server ZoneManager

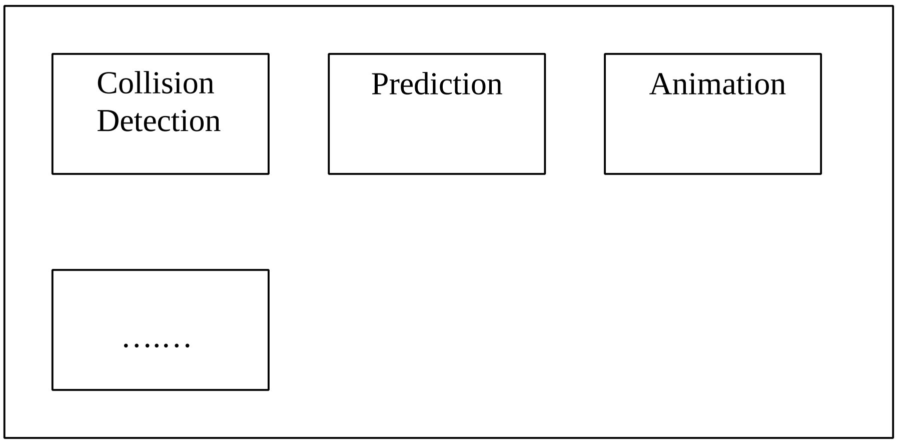
Figure 5.1 Platform Architecture at the Server and the Client

Services is an encapsulation of all the services offered by the other client simulation platform modules. This whole constitutes most of the runtime infrastructure of the client platform. Services offers a common interface (API) that allows the application objects to use the services offered by the different modules. It also provides a few simple services itself, used primarily by the other modules of the platform.

Some of the services are:

ObjectFactory is responsible for creating new objects as they enter the simulation world, including the pilot replica of the client’s avatar. Its major use is by the EventDispatcher, to create local drones when PDUs addressed to unknown recipients arrive.

Other Modules: The services grouped as *other modules* in the diagram can be regarded less as actives service objects, and more as utility libraries used by the simulation objects to uniformly implement various common internal functionality in an efficient and uniform way. They are offered as a part of services rather than as classes used or inherited directly by actors in order to allow “plugging in” alternate classes to do the job without having to edit any code in the various actors, or replacing specific class implementations



.

Figure 5.2 Modules Supporting the Behavior of Simulation Objects

5.2.1 Actor Behavior

These services/libraries are the part of the platform most directly affecting actor behavior:

Prediction. This is essentially a factory service for producing instances of various *MotionPredictor* subclasses. These objects encapsulate logging and prediction algorithms that drones can use to extrapolate their behavior between PDUs, and their pilots can thus use to optimize their drone-updates to save bandwidth. Even the navigation system uses to calculate the effects of navigation input between updates.

Predictors share the same simple interface: They have an *addSample* function, which is used to feed it an observation of the real kinetic status of an actor (timestamp, position, orientation, speed, rotation, accelleration) whenever the ‘owner’ sees fit. They also have methods like *getPositionAt* and *getOrientationAt*, which return predictions of what the situation will be like at a specified (usually future) point in time, based on the history of previous samples it has logged. What model/algorithm is used to calculate the prediction, and how much history information it actually stores for this purpose is internal and predictor-specific. There are several different typical predictors available as part of the platform, representing different common physical behavior models (a ball with a given position, velocity and rotation will proceed differently from a car with the same starting conditions). More can be added as required by the application.

Collision detection

Another task that many actors have in common is discovering and reacting to collisions with other geometry, both the scenery and other actors. As far as scenery is concerned, one could exploit the collision detection in the player, since he scenery is static and reactions to it relatively simple. The current prototype does its own detection here too; see section 5.2.4. In either case it is encapsulated as a service, so actors need not care how it's done.

Collisions with other actors are more complex, since both objects may be moving. What to do to avoid collision, or how resolve the results when collisions do occur, is less obvious. Again, the strategy employed can be application specific, but certain basic behaviors are included with the platform, such as stopping dead when colliding, avoiding collisions by employing a forced right hand rule, or even *allowing* users to pass through others (useful for very crowded areas).

Both detectors are used by calling their *check* methods with the endpoints of the proposed movement (typically each camera update). The return value indicates if the move is legal or not, and optionally you can get a suggested alternate destination that is legal. Thus a movement that would have crossed a wall can stop at the wall in stead, and a mutual attempt of two actors to walk through each other can be resolved by forcing them both to sidestep.

5.2.2 Networked Interaction

Event Dispatcher

This service module is concerned with delivering the incoming events to all interested objects. Each incoming event is tagged with the name of the zone where it has been originated, i.e., the home zone of its originator; it may, in addition, be targeted to other zones as well.

Upon receiving an event, the EventDispatcher finds out about all objects interested in the event. This is done using the ZoneManager described below. The EventDispatcher then dispatches the event to all those objects.

A corresponding module exists also in the server platform. Both the client and the server versions of this module perform the same task of dispatching an event to the targeted objects, but they differ in a subtle way. The client EventDispatcher distributes the incoming events to the local objects, i.e., in the local domain, while the server EventDispatcher dispatches the events over the network, in the global domain. The former therefore needs the local addresses of the objects targeted by the event while the latter needs network addresses. The client EventDispatcher at each of the network addresses receives the event, which it further distributes to the targeted local objects.

This module has two interfaces: one used by the local objects to dispatch events and one used by the network module for delivering events.

The EventDispatcher processes two types of events: system-level and application-level events. The former type allows for communication between the server's event-distribution modules (objects) and the clients' simulation platforms. For example, a simulation object joining a zone uses the API offered by the Services module to do so. This service is supported by the ZoneManager which tells the target zone to add a new member to its membership. If the target zone had no member from before, this client is not registered with the server ZoneManager for receiving events from that zone. It therefore generates a system-level event to be dispatched to the server event-distribution module telling the server ZoneManager to add the network address of the client to the membership of the target zone. The system-level events are all defined by the MADISON platform.

ZoneManager

The client ZoneManager module manages the zones defined by the simulation application. Each zone is implemented as an object that keeps track of the IDs of its member actors. The class *Zone* is defined by the MADISON platform to be used as the superclass of all zone objects, i.e., the zones defined in an application must inherit from it and define its methods. The zone objects are always accessed through the ZoneManager.

Joining and leaving zones can be done either directly by application objects via the Services module API or indirectly, in an automated way, by the zone objects themselves. To achieve the latter alternative, each zone object has a method for checking whether an object qualifies for membership in the zone. If an object passes the check, it is automatically added as a member to the zone, if not already there, and if not, it is removed from the zone membership. However, this method is defined by the application for each of the zones it defines and will usually involve checking values of some global attributes of an object. This latter alternative is not implemented.

The main structure of the ZoneManager is a table mapping zone names to the corresponding objects, and the main structure of a zone object is a list of its members.

PDU hierarchy

The MADISON platform defines a class hierarchy, rooted at the class *BasePdu*, for sending synchronization messages between a pilot and its drones.

Pilots notify their drones of their state-changes by sending them state update events, and inform them of their actions by relaying independent action or directed action events. Upon receiving these events, the drones either synchronize their states with that of their pilot, or perform the action indicated by the received action event.

In the case of actions, to ensure that the drones perform the action in the same context as the pilot, each action event also carries all the information of a state update event. The drones first update their states and then carry out the action in the updated state. The overhead of carrying a little extra state information in each action is judged to be negligible with respect to the overhead involved in correcting the effects of an action carried out in a stale state.

The events are exchanged between the clients via the Server's EventDispatcher. In the client platform the PDUDispatcher and the network classes are responsible for the sending and receiving of the events. The events are implemented by the *Pdu* class hierarchy depicted in Figure 5.3. The PDU classes are described in more detail in [6].

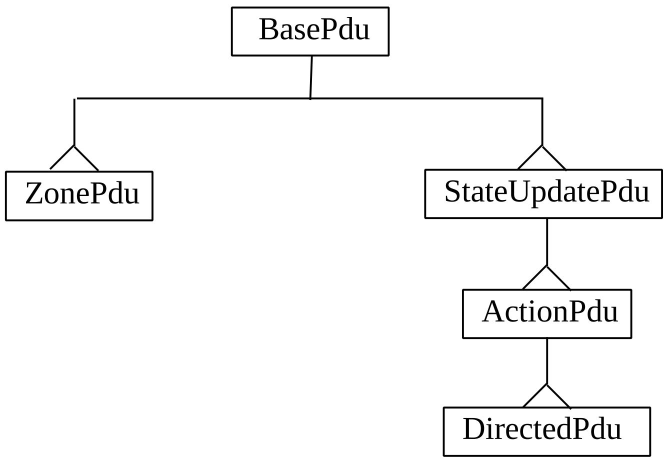


Figure 5.3 The event class hierarchy

5.2.3 Local Interaction

Object Registry

The ObjectRegistry module holds information about all actors local to a client. The main purpose of this registry is to map the unique identifier of each of the locally represented actors, to the actual java object reference of the local instance. In addition, it supports checking on some of the attributes of an actor, e.g., whether an actor is a pilot, and provides a facility to search registered actors using a caller-supplied test, encapsulated in a Predicate object, returning the IDs of the set of actors that pass the test, or the one that scores best on the test.

Actors can obtain the local addresses (java object references) of other actors residing on the same client by using the services of the ObjectRegistry module, as described at the start of this chapter.

There are two mechanisms for obtaining the local address of actors: one used to obtain the local address of a specific actor and the other used for obtaining the local address of an actor (or a set of actors) satisfying a given criterion. In the former mechanism the unique ID of an actor, e.g., obtained from an event, is provided to the ObjectRegistry that maps it to the local address for that actor.

Predicates

In the latter mechanism a test object, or *Predicate*, is provided to the ObjectRegistry, which applies the test to all registered actors and returns the addresses of actors passing the test. The predicate must implement the interface *FuzzyActorPredicate*, defined by the MADISON framework. The actual contents of the tests are application specific and may use any information at their disposal to make their evaluation, including combining other predicates in logical expressions (‘and’, ‘or’, ‘not’ etc.), or any other calculations on them. As the name indicates, the predicates are based on fuzzy logic: the result of a test is a fraction between 0 and 1, indicating to which *degree* the test was successful. Predicates do not have to exploit this, they can opt to return only boundary values (0 or 1) indicating failure or success respectively. Under such circumstances the fuzzy logic operators will behave as normal Boolean ones. But the option is there for actor logics to work with concepts like “close” or “facing me” (and logical operations on them) as continuous-scale values and meaningfully compare “scores”.

Actors

Moreover, all actors in an application must implement the *IActor* interface, defined by the MADISON framework. It defines the capabilities a valid actor must implement, among them *processAction(IAction action)*, which is the entry point for all inter-actor actions. Often it will also be natural to base specific variants on a class from the hierarchy of stock sub-interfaces and their utility implementations, and only override the parts that need to be specialized. The framework includes utility Actors from the absolute-minimal *BasicActor*, down to *BasicAvatar,* which is a complete generic avatar with default implementations of typical Actions.

Actions

Similarly, local actions are supported by classes and interfaces defined in the MADISON framework. All actions are represented by application defined classes, but they must, as mentioned earlier, implement the interface *IAction* defined by the framework, and will often inherit basic functionality from utility implementations, which are also part of the framework.

5.2.4 Client-User I/O

The client java application is responsible for retrieving, interpreting and affecting all user input. Input can originate directly from external devices (mouse, keyboard, gamepad, remote control, etc), or indirectly via the MPEG-4 player, treating event sources in the scene as an input device.

Pluggable ‘driver’ modules in the platform translate input from these various sources to uniform set of events (forward at rate V, rotate at rate R, button 1, button 2, type-in “A” etc.) that the actors can relate to in the same way, regardless of what type of device the actual source is.

In order to obtain the necessary level of control, especially to perform effective predictions, it was for several reasons (current limitations in EAI, MPEG-4, and the available implementations) necessary to drive user navigation ‘natively’ from the Madison java code, rather than exploiting the built-in navigation in the player, which would otherwise have been natural.

This in turn means that the Madison platform must also handle its own collision detection between the user avatar and the world geometry. To this end, the platform provides a *collider* module used to preemptively limit user movements. It is intentionally kept as simple as possible, operating on explicitly defined 2D collision boundaries rather than all geometry. In our experience it works very well within its constraints: as long as the floor plan within a zone is flat, and avatars are earth-bound. The extra processor load is small, anyway balanced by disabling the now redundant collision-detection in the player (more optimized but also vastly more complex) so system performance does not suffer. Collision boundaries for a scene are quickly and easily designed in the MadEdit tool (see chapter 6).

Recent signals from the Web3D consortium indicate that they are aware of the limitations that led us to this duplication of functionality, and intend to address them in their upcoming successor to VRML. The proposed changes, most likely based on Blaxxun’s input framework extension to VRML, appear to be suitable for enabling a return to using the built-in player navigation with its more advanced 3D collision detection. It is likely that the extensions will find their way into future MPEG-4 profiles, since the consortium is the creative force behind all the relevant parts of the existing MPEG-4 standard, and they cooperate closely with MPEG.

5.3 The MADISON Server Platform

The architecture at the Server is different from that of the client in that it consists of two main components:

- a complete client platform, as described in the previous section, called **Server-side Client**, and
- a module handling the server specific services, developed by Archetypon.

The Server-side Client is a complete simulation client hosting the pilot replicas of all bot objects, the drone replicas of all the avatars, and some special bot objects providing common control of the application. From the simulation and platform point of view this client is treated just as any other client.

6 Tools implementation

Two authoring tools for world authoring have been produced. They are rudimentary, but already very useful, and draw up the main structure of an extensible system that should scale well into a more complete production environment, given some extra development.

A third tool for partial generation of avatar and other actor code is the next logical step, but the unexpected demands of the generic client platform code had to be given priority over this.

6.1 The World Integrator

The first tool is a batch program for automatically adapting and consolidating content from different free-standing VRML sources into a monolithic file suitable for compiling directly to .mp4 format. It is suitable for use in scripts or makefiles typically used for building the complete application.

Keeping conceptually different portions of the world in separate source files and editing them separately significantly increases maintainability of the components, and speeds the development cycle. It also imparts much greater flexibility in the production process, making it more suitable for exploiting various editing tools (both third-party and our own) for creating and editing the different components. Having this stitching-together done automatically as a quick and effortless part of the build process eliminates a slow and error-prone manual phase in each update of the scene.

6.2 MadEdit

The second tool is an interactive graphical world editor, dubbed “MadEdit”, consisting of an applet working in tandem with a VRML player (much like the client itself). The tool allows loading the world geometry (typically produced in an external design package such as 3Dstudio) as a “backdrop”, and then visually adding and editing Madison-specific info, in-place in the 3D world. The added information can then be exported as generated files containing the appropriate VRML and java code fragments, ready for use in compiling the final world and application.

So far MadEdit supports editing collision boundaries, but it is suitable for extending to cover various other world-related features that have so far been defined “manually”. Examples are zone definitions and boundaries, teleports between them, java embedding, and tailored makefiles for combining the generated content with standard templates and hand-coded files into the final application.

MadEdit is intended to form the basis for a more complete package covering all world-related authoring and assembly tasks in Madison that are suitable for automation. (Except actual 3D modelling; this is best left to existing third-party tools, which is an advanced industry in its own right).

References

[1] ISO/IEC. *Coding of Moving Picture and Audio*. JTC1/SC29/WG11 N2739 subpart 3, in Recommendation 14496-1 (MPEG-4 version 2 MPEG-J), Seoul, Korea, 1999.

[2] ISO/IEC. *The virtual Reality Modeling Language (VRML97) – Part 2: External authoring interface.* Committee Draft International Standard 14772-2:1997

[3] MADISON. *MADISON architecture*. Deliverable D1, MADISON, April 2000.

[4] MADISON. *Behaviour Definition Toolset – Interim Implementation*. Deliverable D5b, MADISON, February 2001.

[5] MADISON. *Simulation Server Implementation*. Deliverable D6c, MADSION, September 2001.