MicroCosm^{TM} Object Protocol

how MicroCosm simulates the world using objects
and messages betwixt players and host
by
Chip Morningstar

The Protocols of The Elders of MicroCosm

Lucasfilm Ltd. Games Division
February 18, 1986

Introduction

This document presents the object model used to simulate reality in the **MicroCosm** system together with the communications protocol used to realize this model. Understanding the object model is the key to understanding the entire operation of the system. Implementing the protocol is the key to making the system actually work.

This document does not discuss the details of actually getting bits from one place to another (the so called ‘‘link level’’ protocol). We presume that these problems are already taken care of by the system in which **MicroCosm** is embedded. Rather, this document concerns the remote-procedure-call conventions for manipulating objects and requesting information.

The Object Model

The **MicroCosm** world model is object based. Everything in the world is an ‘‘object’’ — a small unit of program and data that simulates a particular entity in the universe. For example, avatars, trees, automobiles, walls and doors are all objects in **MicroCosm**. An avatar’s actions (and thus a player’s actions) are expressed entirely in terms of objects.

Each object has a *home system part* and a *host system part*. Each of these parts in turn consists of a *data section* and a *code section*. The code section is an executable program fragment that simulates the object’s behavior. The data section is simply a chunk of memory that its respective code section is entitled to manipulate. The contents of the data section determine the object’s *state*. The host system part and the home system part may (and usually will) contain duplicate information in their respective data sections. In such a case, the host system part is considered the ‘‘master’’ part. Its data section is always considered to represent the ‘‘true’’ state of the object, no matter what the state represented by the home system might be. This is because the home system representation of an object is not necessarily unique: more than one home system could be maintaining its own version of that object. Furthermore, the host system’s representation is the only one which can be trusted to remain ‘‘honest’’.

One might ask, ‘‘Why have the home system keep such a representation at all? Why not just keep it all in the host all the time?’’. There are two reasons. First, we don’t want to keep the host busy just answering requests for state information, let alone make the player wait while we transmit this information (at 300 baud!) every time it’s needed. Second, many objects possess ‘‘secondary’’ state information that is not functionally relevant to the rest of the world, but which is necessary or helpful to the home system in displaying the object on the screen or in processing player input.

In order for the player to see a model of the world that is accurate, however, there must be constant exchange of information between the home system and the host. This is because the world is not static. Change is happening all the time. The host must be able to update its internal representation of an object according to the actions that the object participates in. For this to happen, the player’s system must inform the host of the player’s actions as regards that object. Similarly, the player’s system must be kept apprised of actions taken by the host that are the result of other players' behavior as regards that object.

Requests and Responses

We thus come to **The Prime Directive of MicroCosm Object Modeling**: *The host will take no unsolicited actions. All behavior originates with the players. All object state changes in the host are the direct results of player requests.*

The communications protocol linking the home system and the host is *request based*. That is, all communications canonically take this form: 1) the home system realizes that it needs some information from the host that it does not have or that it needs some action taken on its behalf by the host; 2) the home system transmits to the host a *request message* asking for the necessary information or action; 3) the host processes this request somehow; 4) the host transmits back to the home system a *response message* providing the information or indicating the results of the action. If it were a just world (the real world I mean, not the **MicroCosm** world!) this would be all that there is to it. However, technical and structural considerations force the following qualifications to the description of the canonical host-home exchange:

- *The response can be empty.* The home system can send a request that the host receives and processes, but which the host does not respond to. Obviously there are two ways that this can happen. The first is if something goes wrong: a message gets garbled in transmission or lost in the packet switching network. The other is by design: the request is such that it simply doesn't need a response. The only way for the home system to distinguish between the two is its own expectations. It has to know whether it is expecting a response. If it is expecting a response and doesn't get one after some time passes, then it knows something is wrong. If it isn't expecting a response then it never cares that one never comes. This means that we have to construct the home system to distinguish between these two types of requests. We do intend to use responseless requests wherever possible to cut down on communications overhead. There is an added complication that a request may be one with no expected response, but if something goes wrong a response may come anyway (perhaps, ‘‘hey, that one got garbled, send it again’’). The home system must be prepared to deal with a response even though it isn't expecting one — it can't just send a request and then immediately forget it.
- *The request can be empty.* The host can send a response to a request that the home system never issued. Again, there are two ways this can happen: by error or by design. We eliminate the possibility of errors of this sort by constructing the host to always send response messages to the right places and then hoping that our host software doesn't have the wrong sorts of bugs in it. Thus, all response messages can be assumed to be ‘‘real’’. ‘‘What about the ones with no requests to go with them?’’, you ask, ‘‘Isn't that a little odd?’’. We actually have a use for such things. They are called *asynchronous notification messages* and serve to inform the home system about external actions that are relevant to it. An asynchronous notification is the result of a request that some *other* player made which affects you. This other player's request initiated some action in some object in the host, and the code simulating that object in the host sent you a message about it. Such messages arrive at the home system as responses to the *phantom request* which the home system must be prepared to pretend that it sent even though it didn't really. This phantom request can be thought of as a request to ‘‘tell me something interesting or important’’. The standard message protocol allows the host to be very specific about what the home system should know or do as the result of an asynchronous notification.
- *The communications medium is not ideal.* In f act, it's downright awful. What we have are 300 baud modems connected to a packet switching network. 300 baud means the time required to transmit information is substantial, so we must try to minimize the amount of transmission we need. The packet switching network means that there may be substantial delays (50 milliseconds to 5 seconds, though typically 100 to 200 milliseconds) between request and response, even if the host responds instantaneously. If the player is doing lots of things, we don't want to keep him waiting as we send out one request, wait for the response, send out another, wait for the response to *it*, send out a third, etc. What we want to do is send a stream of requests continuously while concurrently processing a stream of responses. In other words, we can send out a request without having in hand the responses to previous ones. This means that the home system must keep a list of pending requests. Probably this list will be small (say 4 to 8 items) and when it is full we actually do have to wait for some response before sending out anything further. This smooths out the player interaction a great deal, but adds

considerable complication to the home system software. Things we will have to worry about include: requests that depend on one another and therefore really do have to be serialized, requests that get lost on the way to the host and therefore never get responses, and responses that get lost on the way back from the host and therefore the home system gets confused about whether certain actions were actually taken. So it goes.

Messages

As discussed above, all actions in **MicroCosm** are expressed in terms of the behavior of objects. A communication between the home system and the host system is actually a communication between the home-resident part of an object and the matching host-resident part *of the same object*. In other words, all communications are internal to particular objects. Although each object has its own code and data space, all the objects must share a single common communications channel to link their halves.

Thus, a message consists of three parts: 1) a *header byte* for sorting out asynchronous messages, 2) an *object identifier*, indicating the object whose internal communication the message is carrying and 3) the communication itself. The communication itself in turn consists of a *request number* and, optionally, some *request parameters*. The request number indicates what action or information one half of an object wishes from its counterpart. The request parameters, if there are any, provide additional information whose interpretation depends on the request number.

Message Format

The format of a message is thus:

Byte	**Contents**
0	header byte
1	object identifier
2	request number
3-N	parameters

This format does not address lower level protocols which may have to wrap additional information around this packet to insure that the bits travel successfully from one place to another. These will undoubtedly tack additional bytes on the front or rear of the message packet. Such link-level protocols are outside the scope of this document.

A few comments are in order.

1. Bidirectionality

This protocol serves messages traveling in both directions: requests *and* responses. The home systems assign *sequence numbers* (contained in the header byte) to outgoing requests however they like (probably using a ring counter, but I don't *think* it really matters). Each home system has its own set of sequence numbers — the numbers one player's system uses have no effect on any other player's system. A response message contains the sequence number of the request that it is a response to. A special sequence number is reserved for the phantom request, that is, for asynchronous notification messages.

2. Header bytes

The header byte has the following form:

```
01c0xxxx
```

where `c` is the *message continued* flag bit. A `0` bit means that this is the final packet of a message while a `1` bit means that more packets follow. The`xxxx` is a sequence number in the range 0 to 15 (it wraps around at 15). The resulting codes are printable ASCII:

`@ABCDEFGHIJKLMNO`	for "not continued" packets
`‘abcdefghijklmno`	for "continued" packets

The special header byte for responses to the phantom request (asynchronous notification messages) uses the sequence number 26 (decimal) so that the header byte is ‘‘Z’’ or ‘‘z’’.

The sequence number is only four bits. This means that there may be no more than 16 unrespondent requests pending at any given time. In actual practice, this limit may be even smaller, say 4 or 8. Even with the smaller limit, the full space of 16 request numbers should be available to give greater time separation between uses of any one number. Such separation may help in error recovery when things get *really* messed up.

3. Object identifiers

The object identifier is a single byte. This will be the byte identifier for objects which the host established when the player’s avatar entered the region that it currently occupies.

4. Request numbers

The request number is a single byte. This means that there can be no more than 256 distinct requests that you can give to a particular object. That’s a hard limit and the objects themselves will have to be designed with this in mind. In practice this should not be an important constraint, because the objects themselves will be quite simple (and thus so will their behavior). In fact, we are setting aside the first 128 request numbers for *generic requests*. Generic requests are requests that all objects in the universe are expected to be able to respond to, and they will all have conventional, pre-assigned meanings that are the same across objects. These requests will include things like ‘‘transmit the image set describing this object’’, ‘‘transmit a complete copy of this object’s current state data’’, ‘‘send me this object’s noid’’, and so on. The second 128 request numbers are reserved for object-specific requests. These are requests that relate to the particular characteristics of the object, such as ‘‘open this object’’ (directed at an object which is a container), ‘‘turn this object color X’’ (directed at an object which can change color), or ‘‘shoot at player Y’’ (directed at a weapon object).

There are actually two interpretations of the request numbers for any given object: one for requests and one for responses. Most of the above discussion concerns requests. The request numbers in responses also must be dealt with. In most response messages, a request number R simply means ‘‘response to a request of type R’’. The relevant information, if any, is contained in the request parameters, and the object behavior code associated with the object in the home system should know what to do. Sometimes, though, a response contains a different request number than the corresponding request. This means that the object behavior code is being directed in some fashion by the host, and the home system must comply appropriately. Consistent with the rest of the object model, the appropriate action in such a case is a function of the object and the request number.

5. Request parameters

These parameters are provided because some requests require arguments. For example, moving an object is accomplished by a ‘‘move’’ request, but you must specify the location to move to in the request parameters. Whether the request parameters are interpreted as single byte values, character strings, noids, multiple precision numbers, or anything else, and what these values in turn *mean*, depends entirely on the object to which the request is directed and what the request is. There may be any number of bytes of request parameter information, up to the limits imposed by the link-level communications protocol (i.e., maximum packet length minus the number of overhead bytes in a message).