

# MASSIVE-3 / HIVEK Application Programming: Behaviours

Chris Greenhalgh, 7<sup>th</sup> July 1999.

## Introduction

A MASSIVE-3 application is normally created as a complete stand-alone application, which uses the system's standard class-based API. Behaviours provide a more modular and script-friendly method of introducing program code into an Environment. They also support dynamic use of distributed behaviours, i.e. blocks of code which are automatically executed in other processes which are sharing the same Environment. Distributed behaviours are useful for subjectivity and run-time reconfiguration (e.g. movement rate limiting and interpolation is implemented using distributed behaviours).

## Instantiating a Behaviour

A behaviour is added to an environment, implicitly, by adding a data item of type BehaviourData to the Environment. The BehaviourData item has three attributes:

- name – a string, the name of the Behaviour to be used;
- value – a string, initial configuration information for the Behaviour; and
- context – a set of bit flags, indicating which Environment replicas should create the Behaviour.

The possible context bit flags are:

- AGENT\_RESPONSIBLE – the Environment replica within which the BehaviourData item was first created;
- AGENT\_MASTER – the Environment replica which is the master for that environment, i.e. normally the main world application (if different from the responsible agent);
- AGENT\_OTHER\_SLAVES – all other Environment replicas, i.e. other clients viewing the same environment; and
- AGENT\_REPLAY – an additional pseudo-context, which (if set) indicates that the Behaviour should still be instantiated, even in the Environment is just a replay of pre-recorded activity. Behaviours which provide normal interaction and application functionality should NOT have AGENT\_REPLAY set – they should not be active in a replay because they would try to respond to things when they have already responded (in the original Environment). The only Behaviours which should be active on replay are ones which provide local behaviours which do not generate events (e.g. the interpolator).

For example, the standard rate limiting and interpolation behaviours are specified for a user's embodiment as follows:

```
env->addNewBehaviour(&topId, RATE_LIMIT_FILTER, moveRateBuf,  
                    AGENT_RESPONSIBLE);  
env->addNewBehaviour(&topId, INTERPOLATE_FILTER, moveRateBuf,  
                    AGENT_MASTER | AGENT_OTHER_SLAVES | AGENT_REPLAY);
```

Note that the rate limit is only instantiated in the user's own client – it limits the rate at which update events are allowed onto the network. It should not be used in replay because it has already done its job (limiting the number of events captured by the – presumed – central recording).

The interpolator runs on all other process, and generates continuous movements from a sequence of discrete steps. It is also used on replay, since it just causes each process to do local interpolation, which was part of the Environment as originally experienced (but not otherwise reflected in the logged events).

## Deleting a Behaviour

A behaviour is automatically deleted when the corresponding BehaviourData item is removed from the Environment (or the process leaves that Environment). The Behaviour should tidy up on deletion.

## Defining a Behaviour

Two new classes must be created in order to add a new behaviour to a process:

- a behaviour factory class, which subclasses BehaviourFactory, and which is used to instantiate the new behaviour, and
- a behaviour class, which subclasses Behaviour, and which encapsulates the code for the new behaviour.

For example, the standard interpolation behaviour declares the following factory class:

```
class RateLimitFilterFactory : public BehaviourFactory
{
    static RateLimitFilterFactory initialInstance;

    RateLimitFilterFactory();

    // virtual method used to instantiate filter
    virtual Behaviour *createBehaviour(Environment *env,
                                       BehaviourData &info, int context);
};
```

which is defined as follows:

```
RateLimitFilterFactory::RateLimitFilterFactory()
{
    registerFactory(RATE_LIMIT_FILTER, this);
}

// virtual method used to instantiate filter
Behaviour *
RateLimitFilterFactory::createBehaviour(Environment *env,
                                       BehaviourData &info, int context)
{
    return new RateLimitFilter(env, info, context);
}

// static instance
RateLimitFilterFactory RateLimitFilterFactory::initialInstance;
```

Note that there is one built-in static instance. This will be automatically initialised by the C++ run-time when the program is started. The constructor of this initial (and only) instance registers its presence using registerFactory; this allows the factory to be located by name when Behaviours are required.

When a new Behaviour instance is required, the system will locate the named factory and call the virtual createBehaviour method; this creates an instance of the appropriate Behaviour class.

The rate limit behaviour itself is encapsulated in a new subclass of Behaviour, declared as follows:

```
class RateLimitFilter : public Behaviour
{
private:
    // instance-state . . . .
public:
    RateLimitFilter(Environment *env, BehaviourData &info, int context);
    ~RateLimitFilter();

    // instance methods . . . .
};
```

In the constructor, the behaviour can use the normal system API to establish callbacks, e.g. based on time, or changes in the Environment. It must depend on these callbacks to have code executed after its initial creation. The constructor arguments include the BehaviourData item which caused this Behaviour to be instantiated; this gives the Behaviour access to the behaviour specified name, configuration value, defined context, and position in the item hierarchy.

When the corresponding BehaviourData item is removed from the Environment (or the local Environment replica is deleted, e.g. left) then the Behaviour destructor is called, and the Behaviour must completely unregister all callbacks, and free all data.

### **Compiling and Linking Behaviours**

At present, Behaviours must be normally linked with the processes within which they are to be instantiated. So, the programmer must still create custom applications (including the necessary centralised behaviours) and/or custom clients (including the necessary distributed behaviours).

In the future this should change to a scheme supporting dynamic loading (c.f. Java class loading), although this will present additional security considerations. It is expected that the API will be unchanged, although the behaviour naming scheme will be refined (e.g. to a URL-like scheme). In the mean-time, please use behaviour names consisting only of letters, digits, minus and underscore characters (in particular, no spaces, slashes, colons, semi-colons or hashes).