Realizing Flexible Consistency in HIVEK

Chris Greenhalgh, School of Computer Science and IT, University of Nottingham, UK.
Email: cmg@cs.nott.ac.uk

1. Introduction

The HIVE project (UK EPSRC research grants GR/L 01527 and GR/L 02074) is concerned with the construction of a run-time kernel for Collaborative Virtual Environments (CVEs). It is based on work on consistency in CVEs from the University of Reading [1], plus world structuring based on an extended notion of Locales (originally from Spline [2]). HIVEK is implemented in C++, and takes the form of a sharable library which presents a class-based Application Programming Interface (API) to the application programmer. HIVEK provides, first and foremost, a shared data service, designed specifically for CVEs. Around this shared data service it also provides a number of other facilities and sub-systems, including a name service, a simple rendering and interaction toolkit, generic network support classes, object serialisation, and network audio facilities.

2. Agents and Environments

The two most fundamental classes in HIVEK are Agent and Environment. An Agent encapsulates the notion of activity or computation, and each Agent will correspond to exactly one main process or thread within the system. An Environment encapsulates the concept of shared data, and each Environment is approximately a shared (replicated) scene graph. A complete running system will comprise a number of Agents, typically distributed across a number of machines, interacting and communicating via one or more shared Environments. This basic division provides a comprehensive and flexible framework for describing both information and computation.

Environments are fully replicated, i.e. each Agent which joins an Environment obtains a complete local copy of that Environment. Consistency between these replicas is maintained by a multicast-based distribution of updates to all Environment replicas, plus additional constraints, which are considered in more detail in the following sections. An Environment implements a tree of typed data items, which are essentially a (possibly partial) scene graph.

3. Environment API

The Environment class provides a standard API which Agent-based applications can use to access and modify an Environment. When accessing an Environment, an Agent can directly read and iterate over the contents of the local Environment replica. Because Environments are intrinsically passive there will be no changes to the local Environment replica while the Agent is inspecting it (unless the Agent performs them explicitly).

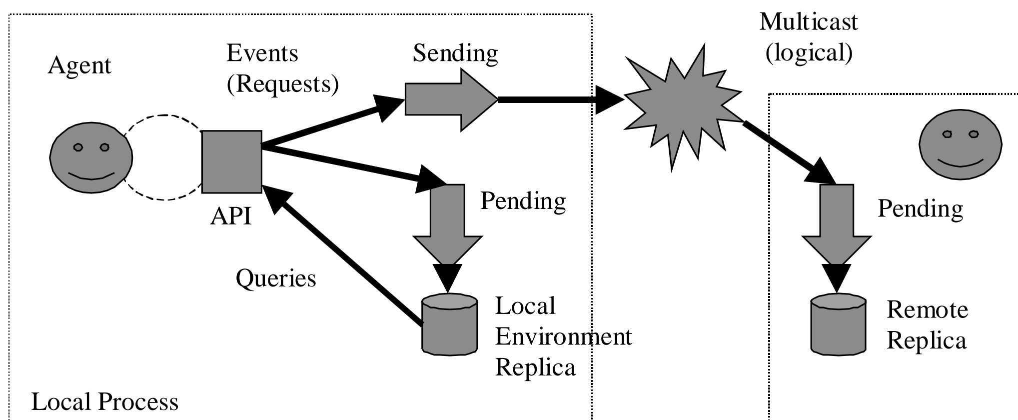

Figure 1: Agents, Environment replicas and messages in HIVEK

An Environment implements an abstract API which supports the creation of new data items in the Environment, and their subsequent update and deletion. When an Agent invokes one of the Environment API methods, for example to update a data item, the abstract API uses the supplied information to create an Event object which represents the request within the system. In normal use each new Event is copied immediately to two event queues which are associated with each Environment replica: a pending queue, which stores Events (originating both locally and

remotely) which have yet to be applied to the local Environment replica, and a sending queue, which stores Events (originating locally) to be communicated to all other replicas of the Environment. In many cases new local Events can pass straight through the pending queue and be applied immediately. However, in some cases the Event may be delayed or otherwise modified by the pending queue. The sending queue is flushed periodically (e.g. once per animation or simulation frame), causing appropriate network messages to be generated. This is illustrated in figure 1.

4. Locking and Ownership

Each item is owned by exactly one Agent at any moment in time, and only the item's owner is allowed to update or delete the item. The owning Agent can transfer ownership of the item to any other Agent (using methods of this same abstract API), which can then update the item, and pass on the ownership again. Any Agent can request ownership of a data item, and this request may be honored or denied by the current owner. Ownership may be requested before it is absolutely required, to avoid subsequent delays in beginning to manipulate the item. For example, in the current user-client ownership of an object is requested as soon as the user's mouse pauses over it in the graphical view, in case they are going to pick it up or move it. Ownership transfers are communicated in the same way as other Events, and are subject to the same potential constraints (see below).

5. Sufficient Causality

Each data item in an Environment has a two-part sequencer associated with it, which allows the causal order of updates to be specified for the item. This also allows the system to distinguish between updates which must be applied reliably (these modify both parts of the sequencer) and updates which potentially can be discarded by the network or any queue (these modify only the second part of the sequencer). For more details of these ideas refer to [1]. Every Event (generated using the Environment API) includes a list of items and minimum sequencer values which must be reached before the Event can be applied, and specifies how the Event changes these sequencer values. By default, each event depends only on the sequencer for the item which it directly modifies, and increases this sequencer appropriately. However, the Agent can specify an arbitrary list of additional items, sequencer values, and changes when it generates the Event. This allows the Agent to specify its own concept of ordering, rather than relying on the communication system to infer a possibly inappropriate ordering (i.e. too strict or too lax).

6. Temporal Constraints

In addition to sequencer value constraints, each Event can also have a minimum delivery time specified. A simple clock synchronisation protocol is used between replicas of an Environment to establish a common "Environment time". An Event will be stalled independently in each pending queue until the Event's minimum delivery time is reached. If other Events are causally dependent on the delayed Event then they will also be delayed, to respect the constraints specified. This allows changes to the Environment to be communicated in advance, so that they occur in a coordinated fashion.

7. Concluding Remarks

One of the key features of HIVEK is the way in which it exposes Events and the sending and pending queues to the Agent. This allows an Agent to directly monitor the activity of these queues, and apply additional constraints or operations to the Events in those queues. For example, within this framework, the sizes of queues provide the Agent with information about Environment activity (the pending queue) and network throughput (the sending queue); this can be used to apply back-pressure to the Agent, causing it to reduce the rate at which it attempts to modify the Environment. Alternatively, the Agent could intercept groups of related messages in the sending or pending queues and replace them with a more concise alternative, for example, removing redundant messages, or constructing summary updates.

References

[1] Roberts, D.J., and Sharkey, P.M. (1997), "Maximising Concurrency and Scalability in a Consistent, Causal, Distributed Virtual Reality System, Whilst Minimising the Effect of Network Delays", Proceedings of the 6th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), June 18-20, 1997, Cambridge, Massachusetts, USA, IEEE Computer Society, ISBN 0-8186-7967-0, pp 161-166.

[2] Barrus, J.W., Waters, R.C., and Anderson, D.B. (1996), "Locales: Supporting Lange Multiuser Virtual Environments", in IEEE Computer Graphics and Applications, 16 (6), Nov., 50-57.