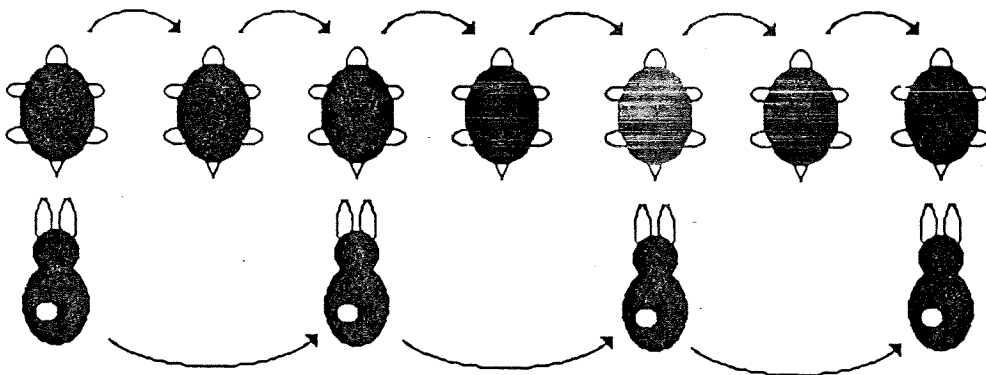
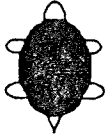



1. A Thought Experiment -- The Tortoise and the Hare

Consider the two simulation objects whose pseudo code is given below. Suppose these objects are running on separate nodes under Time Warp. Since the Hare runs twice as fast as the Tortoise, if each node has infinite memory, then the Tortoise's LVT (local virtual time) would always be half of the Hare's LVT. (This assumes there are no other objects and it takes the same amount of CPU time to run either event section.) How do we get Time Warp to know that these objects are running at different speeds? The next two sections attempt to answer that question. Before jumping on, let's consider this thought experiment in a little more detail.

Obviously this is an artificial simulation and it looks more like two simulations than one. But it isn't as fake as it looks. The current COMMO* program eventually divides itself into two groups (Divisions actually) of non-communicating objects. Also it is quite possible that a simulation would have groups of objects which do not communicate with each other for long periods of time. (Indeed, in the fable, the Tortoise and the Hare were together only at the beginning and end of the race.



<p>TORTOISE OBJECT</p>  <pre>event section { sent a message to myself at time now + 1; }</pre>	<p>HARE OBJECT</p>  <pre>event section { sent a message to myself at time now + 2; }</pre>
---	---

On a more practical note, what happens when memory is limited? Eventually the Hare's node will become filled with messages and states timestamped in the "twilight zone" between the Hare's LVT and GVT (which is roughly equal to the Tortoise's LVT.) The Hare's node is out of memory and flow control will not free any memory. Hence it must wait until the next GVT calculation for memory to become available. As a result the Hare's node is idle for half the CPU time between GVT calculations. (By the way this experiment shows why GVT shouldn't be called every time a node runs out of memory. Otherwise the Hare's node would be constantly calling for GVT.)

Finally we present a variation showing roughly the same type of problem but with two instances of the same object type: "racer". Note again that the Hare runs twice as fast as the Tortoise.

RACEROBJECT

```
event section
{
text = content of the event message;
switch(text){
  case "tortoise": use a CPU unit;/* no break*/
  case "hare": use another CPU unit;
}
sent text to myself at time now + 1;
}
```

2. The Virtual Velocity of an object or node.

There are several possible definitions of virtual velocity. However, each is measured in units of virtual time / real (or CPU) time and is a measure of the change in virtual time over some window of real time. Also of interest is virtual service time, the reciprocal of virtual velocity, which is the amount of real time needed to advance virtual time by one unit. Since virtual time moves in discrete jumps, but real time is closer to continuous, the concept of virtual velocity is perhaps only meaningful over relatively long time periods.

A. Historical Virtual Velocity:

For historical virtual velocity the change in virtual time is the difference between two GVT's (global virtual time computations) and the change in real time is the sum of the processor time that was needed to handle the events in this virtual time window.

NOTE:

(i). If this object or node had no events in this virtual time window, then we are dividing by zero.

(ii). If t_1 and t_2 are the real times at which GVT1 and GVT2 were computed, it is possible that none of the events in this virtual time window got any CPU time in this real time interval. That is the object or node had $LVT > GVT_2$ throughout this real time window. (Hence the name historical.)

(iii). It is also possible that amount of real time used was greater than $t_2 - t_1$.

(iv). Historical virtual velocity is a measure of how much "useful" work was done in the past. It may not reflect what is going on now in the simulation.

(v). It is always non-negative.

B. Local Virtual Velocity:

Local virtual velocity can be measured between any two points of real time. The other difference between historical and local virtual velocity is the way in which the change in virtual time is measured. For local virtual velocity we use the change in LVT (local virtual time) rather than GVT.

NOTE:

(i). If this object or node has no waiting messages, the local virtual velocity could be infinite.

(ii). A rollback could cause local virtual velocity to be negative.

(iii). Local virtual velocity need not represent the rate of which "useful" work is being done.

C. Idealized Virtual Velocity:

This abstraction is the ideal case, messages arrive at regular intervals and it always takes the same CPU time to process them. We also assume that there are no rollbacks, that is we are in a sequential mode. It is easier to define the virtual service time (the reciprocal of virtual velocity).

Let's consider a special case first. Object A receives exactly 1 message every x units of virtual time and it takes exactly y units of real time to process any 1 message. Then object A's virtual service time is y / x and its virtual velocity is x / y .

Unfortunately, the general case is more complex. If object A has two input messages at the same virtual time instant, then the amount of real time needed to process both messages need not be $2*y$. Indeed, if object A was the tortoise of the last section, then amount of real time needed to process 2 (or any other number) of messages is the same as the amount of real time needed to process one message. On the other hand, the commo object in COMMO* must process messages in a pre-given order and hence has to search the list of incoming messages for the next message to process. The current commo object takes real time proportional to i^2 to process i messages that arrive at the same virtual instant. Let's assume that it always takes $y(i)$ units of real time to process i messages that arrive at the same virtual instant.

How often do exactly i messages arrive at the same virtual time instant? Well if we assume that the probability of i messages arriving is $p(i)$, then the average real time needed to advance an object one unit of virtual time is the sum of $p(i) * y(i)$ as i runs from 0 to infinity. Obviously, we are starting to do some analytic queueing theory. However, it seems a little early to attempt this and we will not consider idealized virtual velocity further.

3. Measurement of Historical Virtual Velocity

The following is perhaps a rather naive method for implementing a measure of historical virtual service time (the reciprocal of virtual velocity). Statistics are collected in the "state" structure of the object. A record field called "real_time" is added to the state structure, which holds the amount of real time used to get this object thus far. The current field "aftersave" in the Ocb is used to do a similar measurement in the current Time Warp code. However, aftersave measures only the time the object's code is running, whereas real_time should also include the time used in the object's calls to Time Warp functions like me(), simtime(), etc., since these are a part of the simulation costs and not Time Warp overhead.

A rough estimate of service time is the difference in the real_time fields at two different GVT's divided by difference in the GVT's. Of course the object may not have a saved state at either of the GVT times, thus this estimate will be better over several GVT periods rather than between consecutive GVT's. The service time for a node is just the sum of the service times of the objects on that node. Again this only is a rough measure for each node since the amount of real time between two GVT's can be different for different nodes.

(If "jump forward" is being used, then the real_time field might be more easily implemented as a delta time, i.e. since the last saved state, rather than the absolute time given above.)

4. Time Dialation Objects

It may turn out that Time Warp runs best when all the different nodes have roughly the same local virtual time at all times. Hence we would need an object that "wastes" CPU time in the correct amounts. We call such an object a time dialation object since the faster virtual velocity is at a node, the more the object must waste CPU time to keep the simulation time in "tune".

It is likely that historical virtual velocity is the correct virtual velocity to use in this context. During a GVT calculation a node could sent its "old" historical service time as well as its current LVT. GVT update could include the maximal service time as well as the minimal LVT. The various nodes could slow themselves appropriately.

5. Time-like vs Space-like Messages

It has been suggested that objects that communicate heavily need to be on the same nodes or at least nearby nodes. In general, this is a reasonable rule of thumb. However, there is another dimension to take into account, that of time. If a message from object A to object B has a large $\text{receive_time} - \text{send_time}$ relative to the maximum message delay, then we would expect that rarely would such a message cause rollback even if A and B were as far apart as possible on the hypercube. On the otherhand, object C may rarely sent messages to object D but all these messages have a $\text{receive_time} - \text{send_time}$ of zero. Thus, unless C and D are on the same node (and sometimes if they are), every message could cause a rollback and will if the nodes are at the same LVT.

A time-like message is one which is unlikely to cause rollback, whereas a space-like message is likely to cause rollback. These are only rough ideas which we could try to formalize as follows:

Let:

$x = \text{receive_time} - \text{send_time}$,

$y = \text{delay time to transmit the message (real time)}$,

$z = \text{the virtual velocity of the receiving node}$,

and assume both nodes are at the same LVT at the time the message is sent.

Then the message is:

time-like if $x > y * z$ and

space_like if $x < y * z$.

This is perhaps too refined and using \gg and \ll in place of $>$ and $<$ in the above equations will lead to a more interesting classification.