Modeling and Analysis of Collaborative Virtual Environments by Using Extended Fuzzy-Timing Petri Nets

Y. Zhou, T. Murata, and T. DeFanti

Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607-7053 USA
{yzhou1, murata, tom}@eecs.uic.edu

Abstract. Virtual Reality (VR) systems (such as the CAVE™[1]) generate images in real-time on the basis of the viewer's view in the virtual world, so that the viewer sees a three-dimensional view of a given scene. The concurrency and real-time features in virtual environments systems make them difficult to design, implement and test. Collaborative Virtual Environments (CVEs) make this more complicated by adding network considerations into their designs. CVEs demand high Quality-of-Service (QoS) requirements on the network to maintain natural and real-time interactions among users. By using formal methods to model CVEs and analyze their real-time behavior, we can evaluate the network effects on CVEs and the performance of CVEs. To model temporal uncertainties in CVEs, we propose an extension of Fuzzy-Timing Petri Nets (EFTN) in this paper. We give our EFTN models for the CAVE, the TCP protocol and the NICE (Narrative Immersive Constructionist/Collaborative Environments) project and we analyze the network effects on the NICE and the dynamic performance of NICE.

1 Introduction

Virtual Reality (VR) can be defined as interactive computer graphics that provide viewer-centered perspective, large field of view and stereo. The CAVE™[1] (Cave Automatic Virtual

[1] CAVE™ is a registered trademark of the Regents of the University of Illinois

Environment) ([4], [15], [16]) is a virtual reality environment designed and implemented at the Electronic Visualization Laboratory at the University of Illinois at Chicago. The CAVE, as shown in Fig. 1, is a surround screen, projection based virtual reality environment system. The actual environment is a 10x10x10 foot cube, where images are rear-projected in stereo on 3 walls (front wall, left wall, and right wall), and down-projected onto the floor. (The floor can be considered as floor wall. So there are 4 walls total.) The 4 walls display computer generated stereo images of the virtual world in real-time based on the position and orientation of the user's head and hand in the CAVE. The viewer wears LCD shutter glasses to mediate the stereo images. The viewer's head and hand position and orientation are tracked through sensors on the shutter glasses and on the 'wand' (the CAVE input device). And the viewer can grab and move objects in the virtual world with the wand.

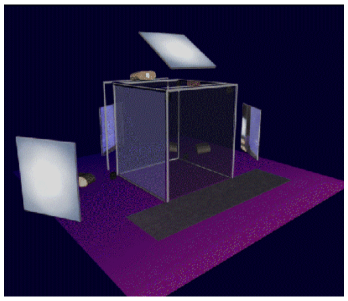


Fig. 1 A picture of the Cave Automatic Virtual Environment (CAVE)

Because of the concurrency and real-time features in virtual environments systems, it is difficult to design, implement and test VRs. Collaborative Virtual Environments (CVE) make this more complicated by adding network considerations. CVEs allow people in remote virtual environments to learn from each other, work together on designing systems, or perform a complex group task together over networks.

The Narrative Immersive Constructionist/Collaborative Environments (NICE) project ([2], [3]) at the Electronic Visualization Laboratory at the University of Illinois at Chicago, is a collaborative learning environment: a virtual garden, where children can do gardening and learning cooperatively. In the NICE, children located in distributed virtual environments (e.g., CAVEs), can take care of a virtual garden together in the center of a virtual island. The children, represented by avatars, collaboratively plant, grow, and pick vegetables and flowers. They make sure that the plants have sufficient water, sunlight, and space to grow, and they keep hungry animals away from sneaking in the garden and eat the plants.

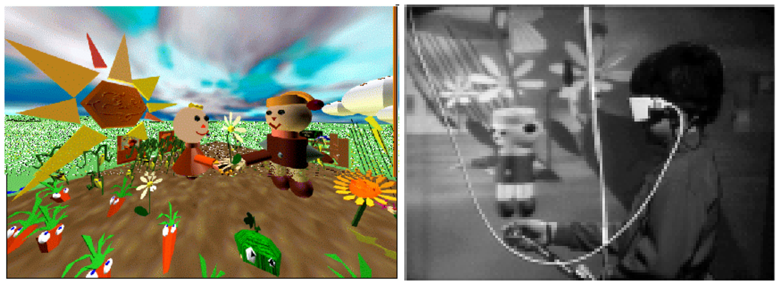


Fig. 2. (a) Jim (an avatar) is handing a flower to Eddie (another avatar); (b) A child is interacting with an avatar in the CAVE.

NICE uses a central server to simulate the garden and maintain consistency across all the participating virtual environments, and a repeater to broadcast all avatar state information. Each virtual environment (VE) sends the local avatar information (the local tracker data) to the repeater by using UDP, and sends the information about the local child's world-changing activities to the central garden by using TCP. The central server receives the world-changing messages from each client, updates the world state and sends the new world information (the information about the garden) to each client by using TCP so that all clients can share the same world information. Meanwhile, the repeater receives each avatar's state information and broadcasts them to all other clients by using UDP. It is very important to draw remote entities in real-time in each VE so that the user will not notice any difference between the local and remote entities in the environment.

CVEs demand high Quality-of-Service (QoS) requirements on the network to maintain natural and real-time interactions among users. QoS refers to the requirements on network latencies and jitters (the variability in network latency). By using formal methods to model CVEs and analyze their real-time behavior, we can evaluate the network effects on CVEs and the performance of CVEs. Petri Nets have rigorous analysis capability and have been shown useful for assuring the reliability and correctness of concurrent systems. In order to model and analyze real-time systems, various timed extensions of Petri Nets have been proposed. However, many real-time systems have temporal uncertainty. For example, the time duration of rendering an image for a wall in CAVE varies on the complexity of the geometric objects in the image, and the network delays in CVEs vary over a large range. To deal with temporal uncertainties in real-time systems, Murata [7] proposed Fuzzy-Timing High-Level Petri Nets (FTHNs) to model time explicitly in terms of fuzzy set theory. FTHNs model temporal uncertainties in real-time systems, and provides possibility distributions of events. So FTHNs can capture all temporal uncertainties in CVEs and they would be suitable models for CVEs.

This paper is organized as follows: Section 2 reviews Fuzzy-Timing Petri Nets and proposes an extension of Fuzzy-Timing Petri Nets (EFTN); Section 3 gives our EFTN models for the CAVE; Section 4 analyzes the dynamic behavior of our EFTN model of the CAVE; Section 5 proposes our EFTN models for the NICE; Section 6 gives the Design/CPN implementation of EFTN models for the NICE; Section 7 discusses the simulation results of EFTN models for the TCP protocol and the NICE; Section 8 concludes the paper and gives our future research plan.

2 Fuzzy-Timing Petri Nets and Extended Fuzzy-Timing Petri Nets

The main features of Fuzzy-Timing High-Level Petri Nets (FTHNs) are the following four fuzzy set theoretic functions of time called *fuzzy timestamp, fuzzy enabling time, fuzzy occurrence time* and *fuzzy delay*. A *fuzzy timestamp* $\pi(\tau)$ is associated with each token and each place, and $\pi(\tau)$ is a *fuzzy time function* or *possibility distribution* giving the numerical estimate of the possibility that a particular token arrives at time τ in a particular place. In FTHNs, arcs (t, p) from transitions t to places p are associated with fuzzy delays $d_{tp}(\tau)$. For simplicity, trapezoidal or triangular *possibility distributions* specified by the 4-tuple $(\pi_1, \pi_2, \pi_3, \pi_4)$ as shown in Fig. 3, are used to represent fuzzy time functions.

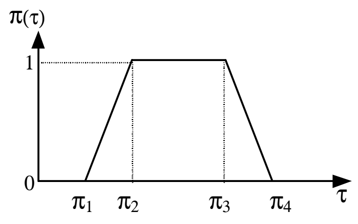
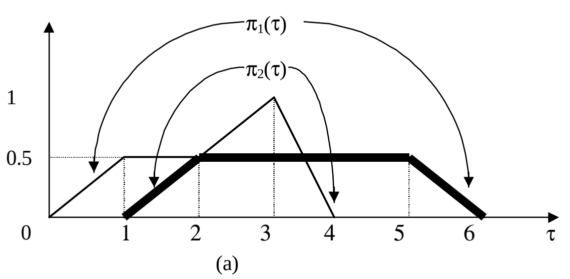


Fig. 3 Trapezoidal *possibility distribution*

The formal definition of FTHNs and the method to compute and update fuzzy enabling time and fuzzy occurrence time when a transition firing occurs, are given in [7]. A Fuzzy-timing Petri Net (FTN) [11] model is an unfolded version of the fuzzy-timing high-level Petri Net (FTHN). We extend FTN by integrating FTN with Merlin's Time Petri Net [5]. We define an Extended Fuzzy-Timing Petri Net (EFTN) model as a 6-tuple (P, T, A, D, FT, CT), where: (P, T, A, D, FT) is a Fuzzy Timing Petri Net, with the default value of $d_{tp}(\tau)$ being (0,0,0,0); CT: T $\rightarrow$ $Q^+ \times Q^+ \times (Q^+ \cup \infty)$ is a mapping from the transition set T to firing intervals with possibility: i.e., each transition is associated with a firing interval $p[\alpha, \beta]$, where the default interval is 1[0, 0] (a transition definitely fires as soon as it is enabled). If a transition t is enabled at time instant τ, t may not fire before time instant $\tau + \alpha$, and t must fire before or at time instant $\tau + \beta$. *Possibility p* $\in$ [0,1]. *p* is 1 if transition t is not in conflict with any other transition. *p* can be less than 1 when we want to assign different chances to transitions in structural conflict. For example, if transition t1 and transition t2 are in structural conflict, t1 fires with 99% chance and t2 fires with 1% chance, we assign $p_1 = 0.99$ and $p_2 = 0.01$. A transition firing itself is an atomic event and takes zero time. (CT is taken from Merlin's Time Petri Net [5].)

Now, in EFTN, the *fuzzy enabling time* $e_t(\tau)$ of transition t is computed by $e_t(\tau) = latest\{\pi_i(\tau), i = 1, 2, ..., n\}$, where *latest* is the operator that constructs the "latest-arrival/lowest-possibility distribution" from n distributions ([7], [11]), as shown in Fig. 4(a).



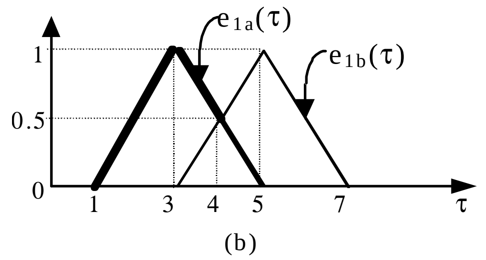


Fig. 4 (a) *latest*{$\pi_1(\tau)$, $\pi_2(\tau)$} shown by heavy line; (b) *earliest*{$e_{1a}(\tau)$, $e_{1b}(\tau)$} is shown by the heavy line.

When there are m transitions enabled with their *fuzzy enabling times*, $e_i(\tau)$, $i = 1, 2, ..., t, ..., m$, and $CT(t_i) = p_i[\alpha_i, \beta_i]$, we compute the *fuzzy occurrence time* $o_t(\tau)$ of transition t whose *fuzzy enabling time* $e_t(\tau)$, as follows: $o_t(\tau) = min\{e_t(\tau) \oplus p_t(\alpha_t, \alpha_t, \beta_t, \beta_t)$, *earliest*$\{e_i(\tau) \oplus p_i (\alpha_i, \alpha_i, \beta_i, \beta_i)$, $i = 1, 2, ..., t, ..., m\}\}$, w /F73d(si)iim

FTHNs provide additional information on partial ordered events in terms of their degrees of possibilities, instead of transforming them into a total ordering. The computations involved in FTHNs are basically repeated additions and comparisons of real numbers and are necessary only for certain finite firing sequences, and need not generate the entire state space. Thus these computations can be done very fast and thus FTHNs are suited for estimating the performance of time-critical systems.

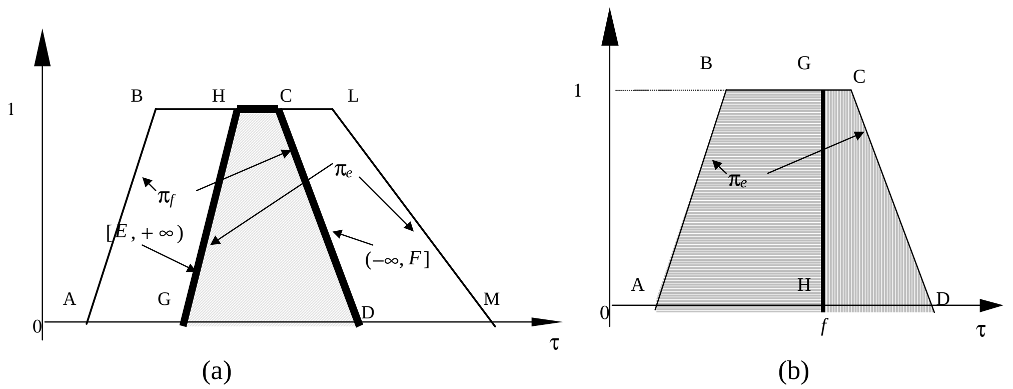


Fig. 5 (a) $[E, F] = [E, +\infty) \cap (-\infty, F]$ is $trapezoid_{GHCD}$ shown by the heavy line, and $[E, F] \cap \pi_e$ is also $trapezoid_{GHCD}$ shown by the shaded area. (b) The part of π_e where $\tau \le f$ is shown by ▤, the part of π_e where $\tau \ge f$ is shown by ▥.

3 EFTN models for the CAVE

The CAVE has the following three main subsystems [4]: (Fig. 7 shows our EFTN model for the CAVE. A timed Petri Net model of the CAVE can be found in [6].)

- *Tracker subsystem*: which obtains data about the position of the viewer's head and hand. Since the viewer wears a head tracker and holds a wand where sensors are located, his position is detected by the tracker operating at 96 HZ sampling frequency. A tracking sample is obtained every 10.4 ms when the monitor signal arises.
- *Main subsystem*: which creates images to be displayed on the walls of the CAVE. There are four graphic pipelines working concurrently. Each of them is used to render the image on one wall. The CAVE implementation uses double buffering between the main subsystem and the display subsystem. While the main subsystem is writing into one buffer, the display subsystem

reads from the other buffer. The buffer swapping is synchronized by a monitor signal at 46 HZ frequency. Once images for all 4 walls have been rendered, a buffer swapping takes place at the leading edge of the next coming monitor signal if the display subsystem is also ready to swap buffers.

- *Image display subsystem*: which draws the images on the four walls. When the drawings of 4 images are all finished, the display subsystem is ready to swap buffers.

4 The Analysis of EFTN models for the CAVE

- *Reduction Rules for EFTN*

In order to analyze EFTN models, we illustrate two reduction rules ([1], [13]) for EFTNs in this section. Our reduction rules can reduce the size of EFTN models and preserve safeness, deadlock and timing properties of EFTN. Applying the two rules shown in Fig. 8 to our EFTN model for the CAVE in Fig. 7, results in a reduced EFTN model as shown in Fig. 9.

- *Behavior of EFTN models for the CAVE*:

In Fig. 9, transitions Swap_and_draw and Swap_Signal_passed are in conflict. Assume that transition Generate_Monitor_signal fires at time π and it is immediately followed by the firing of transition Swap_and_draw, then the next round of image rendering begins with the firing of transition use_data_render. In that round, the 1st monitor signal coming at 20.8 ms will be passed since the image rendering delay is $latest(D_{render\_front}(\tau), D_{render\_left}(\tau), D_{render\_right}(\tau), D_{render\_floor}(\tau)$ = (25.0,37.4,50.0,62.4) ms > 20.8 ms. When the 2nd monitor signal comes at 41.6 ms, the *possibility* that transition Swap_and_draw fires instead of Swap_Signal_passed, is computed as follows:

$$possibility(\, (\pi \oplus latest(D_{render\_front}(\tau), D_{render\_left}(\tau), D_{render\_right}(\tau), D_{render\_floor}(\tau)) \oplus (0,0,0,0)\,)$$
$$<\ (\pi \oplus (41.6,41.6,41.6,41.6) \oplus (\varepsilon 4, \varepsilon 4, \varepsilon 4, \varepsilon 4))\ )$$
$$= possibility(\ (\pi \oplus (25.0,37.4,50.0,62.4) \oplus (0,0,0,0)) < (\pi \oplus (41.6,41.6,41.6,41.6) \oplus (0.2,0.2,0.2,0.2)\,)\ )$$
$$= possibility(\,(25.0,37.4,50.0,62.4) < (41.8,41.8,41.8,41.8)\,)$$

= shaded_area/area_trapizoidal(25.0,37.4,50.0,62.4) = 0.424, as shown in Fig. 6.

If transition Swap_Signal_passed fires, the display subsystem will not begin to draw any new image until the 3rd monitor signal comes. In that case, transition Swap_and_draw can certainly fire when the 3rd monitor signal comes at 62.4 ms, since the image rendering delay is (25.0,37.4,50.0,62.4) ms < 62.4 ms.

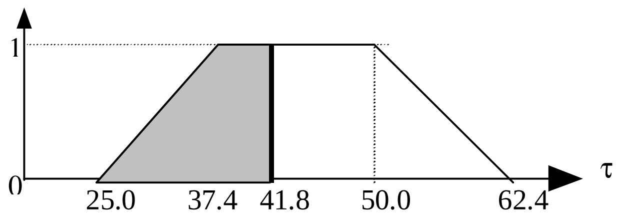


Fig. 6 *Possibility* of transition Swap_and_draw fires instead of Swap_Signal_passed

The *possibility* that transition Swap_and_draw fires instead of Swap_Signal_passed when the 2nd monitor signal comes at 41.6 ms, determines that the delay that the user's movement being reflected on the walls is around 104 ms or around 125 ms: *possibility*(delay ≈ 104 ms) = 0.424, and *possibility*(delay ≈ 125 ms) = 0.576.

We use Design/CPN [14] to simulate our EFTN model for the CAVE. As shown in Fig. 12, a *timestamp* `origt` is attached to each token generated by firing transition Head_Wand_Input. When transition DrawComplete fires, the current time and the original *timestamp* (`origt`) of the token will be recorded into a file. We can calculate the delay by reading the file after the simulation. Our simulation result shows that, the delay is around 104 ms for 1339 times (42.33%), and around 125 ms for 1824 times (57.67%) in total of 3163 times transition DrawComplete firing we recorded. The simulation result is consistent with our possibility analysis.

5 EFTN models for the NICE

The main distributed components of the NICE consist of a garden simulation server, an avatar repeater for avatar state information, and NICE clients ([2], [3]). A NICE client uses an unreliable protocol (UDP) to send avatar information (local tracker data) to the avatar repeater and a reliable socket connection (TCP protocol) to send local avatar’s world-changing messages to the server. The avatar repeater broadcasts avatar state information by using UDP. The NICE server supports the garden simulation, updates the world (graden) once receiving an avatar’s world-changing message, and broadcasts the new world state information to all clients by using TCP.

The Information Request Broker (IRB) is the core of all client and server applications in the NICE. An IRB is an autonomous repository of persistent data that is accessible by a variety of networking interfaces. A key is a handle to a storage location in an IRB's database. Keys are uniquely identified across all IRBs. A local key can initiate and accept multiple linkages to and from other remote IRBs. Any modifications that are made to one key will automatically be propagated to all the other linked keys ([2], [3]).

The garden server is an IRB with two main keys: an incoming message key and an outgoing message key. If the local avatar has any action changing the garden (e.g., plant a tree), the local VE will send a message from the local OUT Key to the server’s IN key by using TCP. Then the garden server updates the world state and sends the new world state information to each client’s IN key via the server’s OUT keys by using TCP. The garden world evolves itself as the plant grows, the weather changes, and animals appear. So the server sends each client the new world state information by using TCP once it updates.

The avatar repeater has a key for each client to hold its avatar-state information. When client1 updates the local screen (swapping-buffer happens), avatar1’s state information will be sent from client1 to the avatar1-state key on the repeater by using UDP. Another client (e.g. avatar2) will get the state of avatar1 by subscribing to the avatar1-state key on the repeater.

In Fig. 10, we give the EFTN model for the garden server, avatar repeater and communication interface of two existing NICE clients communicating with each other, the repeater and the garden server. Each client sends local avatar’s tracker information to the repeater by using UDP and the repeater broadcasts it to all other clients also by using UDP. UDP is a simple unreliable transport layer protocol. By using UDP, the sender just sends out the Protocol Data Unit (PDU) and never retransmits. So we use a transition UDP with fuzzy delay $D_{UDP}(\tau)$ = (50,100,150,200) ms to represent UDP channel in Fig. 10, and we assume the data loss rate of UDP is 1%.

Each TCP transition in Fig. 10 is an abstract of a subnet for TCP protocol. TCP is a reliable and ordered transport layer protocol. One Protocol Data Unit (PDU)'s loss will delay all subsequent PDUs. No subsequent PDU can be delivered to the application layer until that PDU is successfully received. The Design/CPN implementation of our EFTN model for TCP protocol is shown in Fig. 21. And we explain our TCP model in Section 6.

In the NICE, a local VE (e.g., CAVE) will need local avatar state information, remote avatars state information, and world state information to render images. The EFTN model for a CAVE in the NICE as a distributed component is shown in Fig. 11. When an avatar wants to change the world, it usually takes him about 1 second (*possibility distribution* (800,1000,1200,1500) ms) to complete his action. During an avatar's world-changing action, all of his tracker data used for updating local screen, will be sent to the server by using TCP. An avatar may change the world for 2~3 times per minute and the local screen may be updated for 16~24 times per second (16 times/sec if the image rendering delay $\leq$ 41.6 ms each time, 24 times/sec if the image rendering delay is in the interval (41.6, 62.4) ms each time). So, we assume 0.2% of the tracker data used for updating local screen may indicate that local avatar wants to change the garden. (In Fig. 11, when the avatar is not already in an world-changing action, place ChangeWorldOrNot has two output transitions, the *possibility* of firing transition Change is 0.002, and the *possibility* of firing Not_Change is 0.998.) After we put the communication interface and internal structure of distributed CAVEs together, we can analyze the network effects on the NICE and the dynamic performance of the NICE.

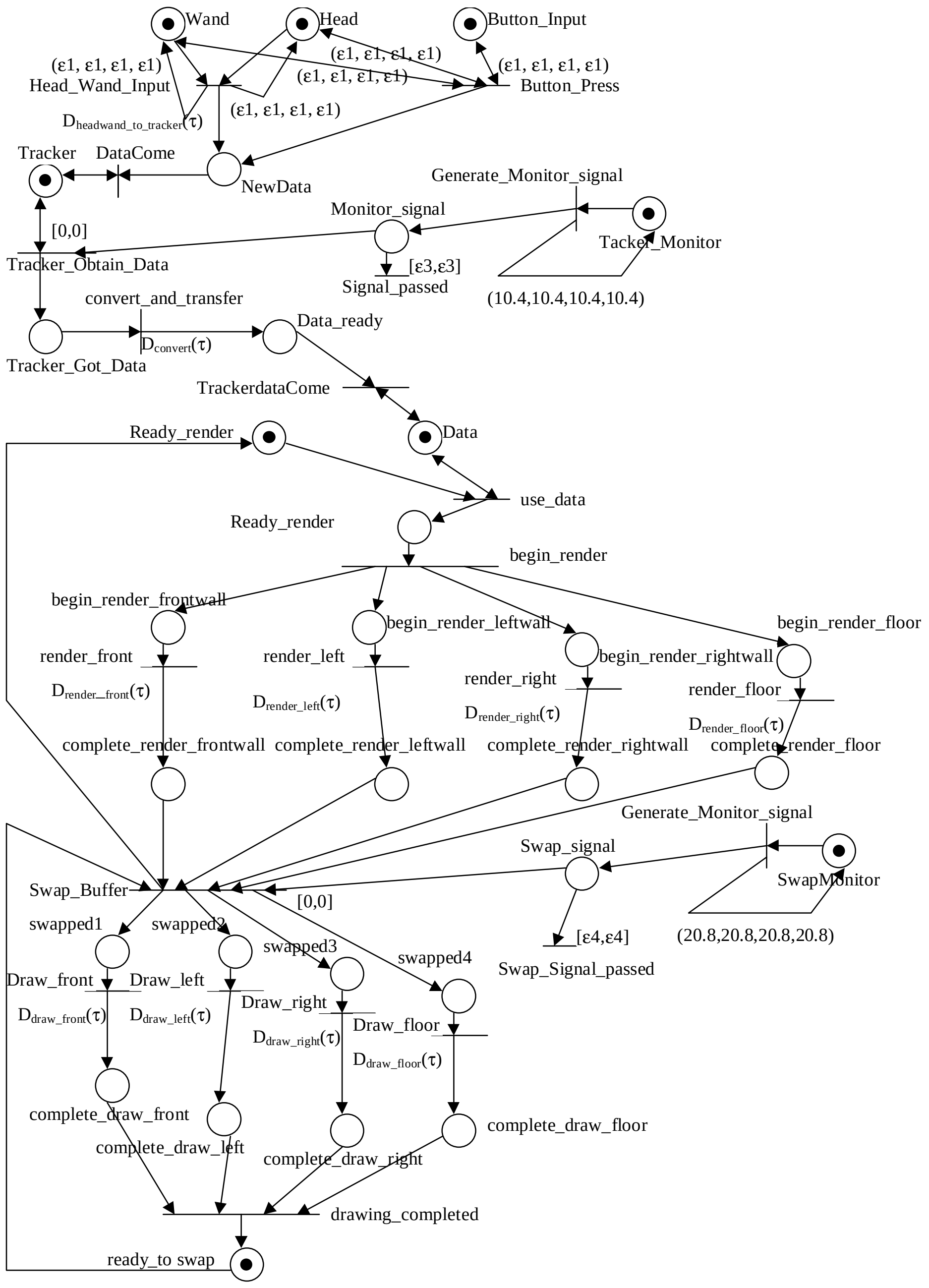


Fig. 7 An EFTN model for the CAVE, where the *timestamp*s of tokens arriving in Head, Wand, Button_Input, TrackerMonitor and SwapMonitor at initial state are $\pi_{Head}(\tau) = \pi_{Wand}(\tau) = \pi_{Button\_Input}(\tau) = (0,0,0,0)$, $\pi_{TrackerMonitor}(\tau) = (10.4,10.4,10.4,10.4)$ ms, and $\pi_{SwapMonitor}(\tau) = (20.8,20.8,20.8,20.8)$ ms. The delay for Head and Wand data arriving at the Tracker is $D_{headwand\_to\_tracker}(\tau) = (50,50,50,50)$ ms. The delay for converting and transferring tracker data to the Unix workstation is $D_{convert}(\tau) = (10,10,10,10)$ ms. $D_{render\_front}(\tau)$, $D_{render\_left}(\tau)$, $D_{render\_right}(\tau)$, and $D_{render\_floor}(\tau)$ are the *fuzzy delays* of rendering images for front wall, left wall, right wall, and the floor. Assume $D_{render\_front}(\tau) = (25.0,37.4,50.0,62.4)$ ms, and $D_{render\_left}(\tau) = D_{render\_right}(\tau) = D_{render\_floor}(\tau) = (10,20,30,35)$ ms, since the image on the front wall is usually more complicated than the ones on other walls. The delay for drawing images on each wall is $D_{draw\_front}(\tau) = D_{draw\_left}(\tau) = D_{draw\_right}(\tau) = D_{draw\_floor}(\tau) = (2,2,2,2)$ ms, and ε3 = ε4 = 0.2 ms (a short time period that a monitor signal lasts).

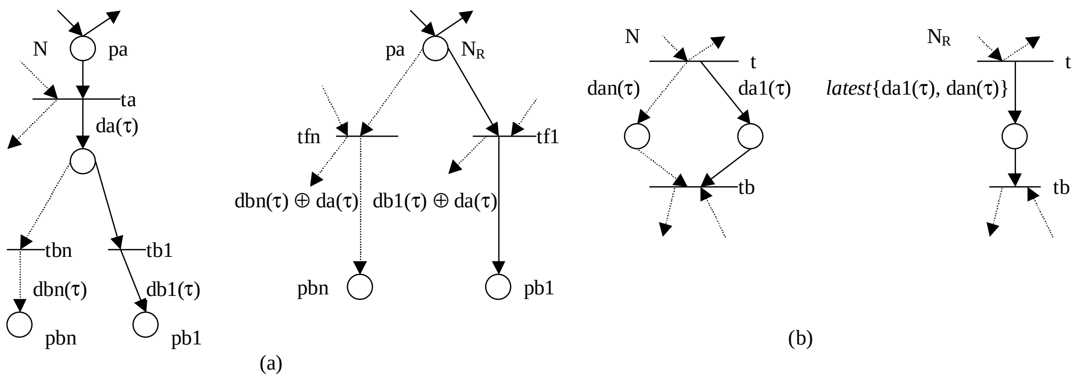


Fig. 8 (a) Post-fusion (post-fuse transition ta with tb1, …, tbn); (b) Parallel fusion of places.

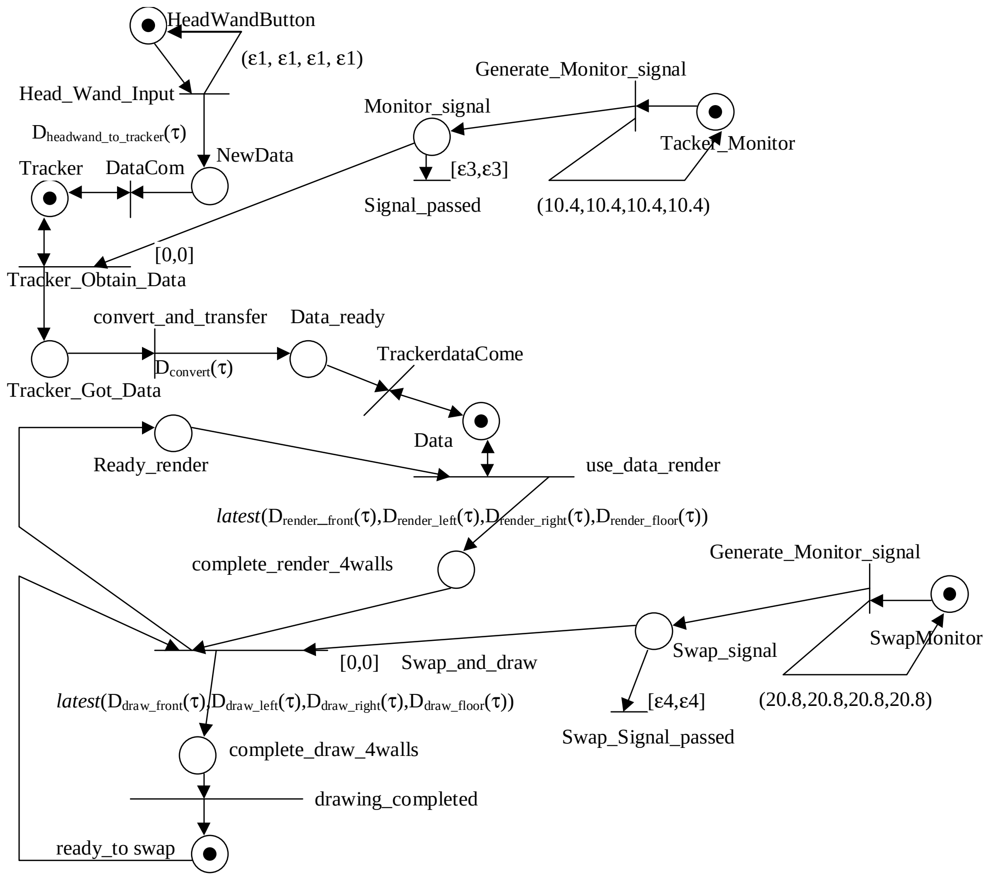


Fig. 9 The reduced EFTN model, where *latest*($D_{render\_front}(\tau)$, $D_{render\_left}(\tau)$, $D_{render\_right}(\tau)$, $D_{render\_floor}(\tau)$) = *latest*((25.0,37.4,50.0,62.4), (10,20,30,35), (10,20,30,35), (10,20,30,35)) = (25.0,37.4,50.0,62.4) ms, and *latest*($D_{draw\_front}(\tau)$, $D_{draw\_left}(\tau)$, $D_{draw\_right}(\tau)$, $D_{draw\_floor}(\tau)$) = (2,2,2,2) ms.

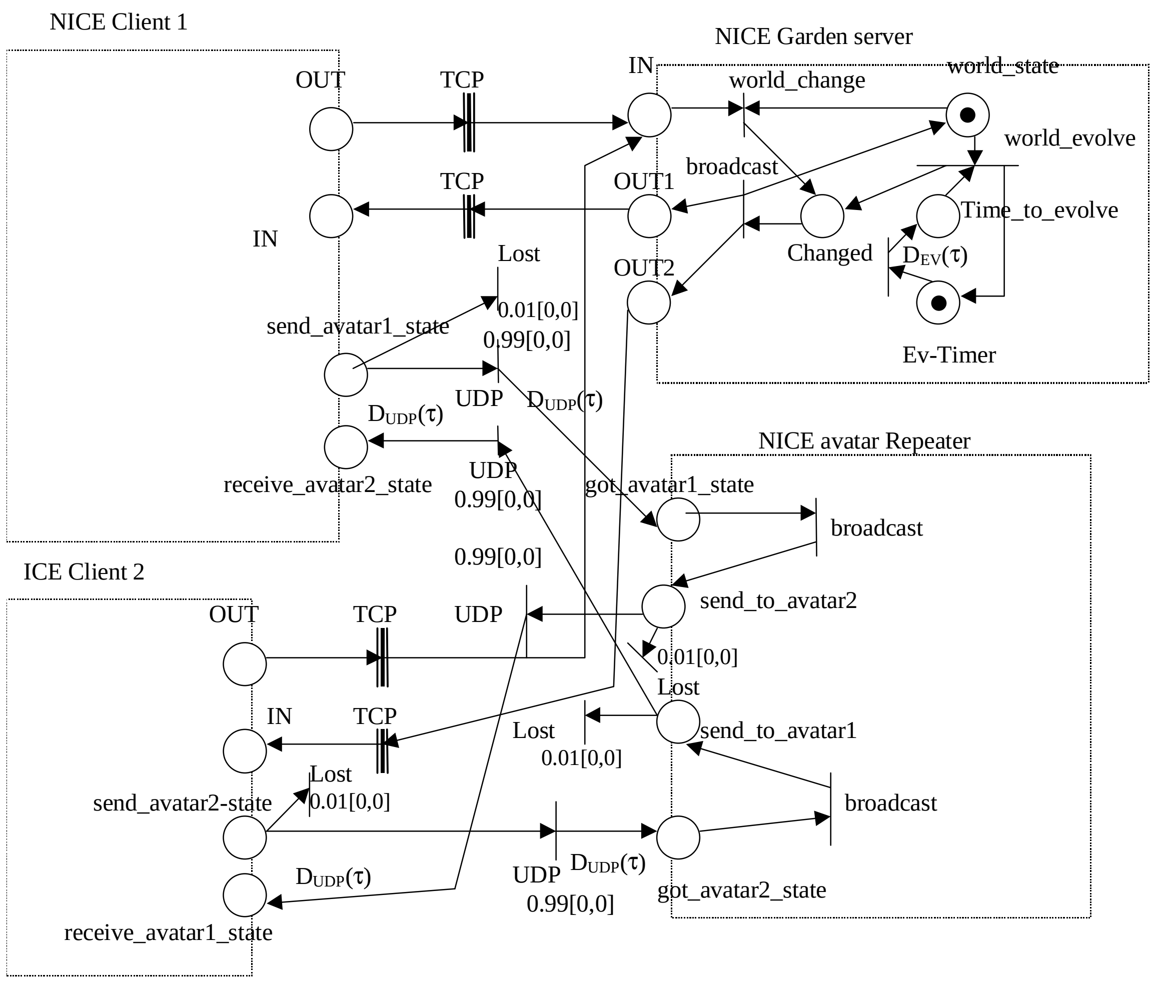


Fig. 10 An EFTN model for 2 existing NICE clients communicating with each other, the repeater and the server, where the delay of UDP channel is $D_{UDP}(\tau)$ = (50,100,150,200) ms, the world evolves in the interval $D_{EV}(\tau)$ = (10,60,180,300) sec.

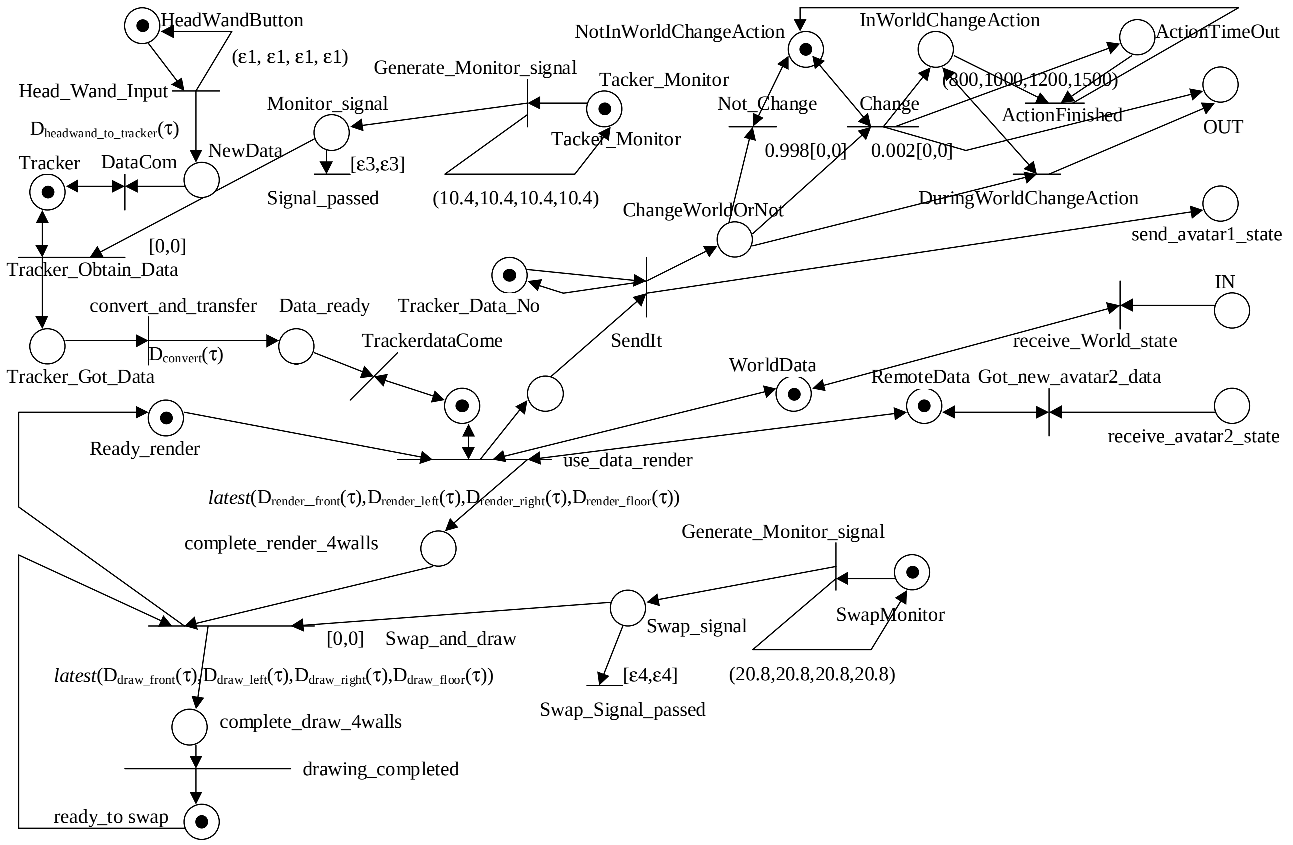


Fig. 11 The EFTN model for a CAVE in NICE as a distributed component

6 Design/CPN implementation

6.1 Global Declaration Node and Fuzzy Time Function

Fig. 15 shows the global declaration node of the Design/CPN implementation for our EFTN models of the NICE. To use Design/CPN for implementing our EFTN models, we need a `function` to generate *fuzzy delays* in *trapezoidal possibility distributions*. Given a *fuzzy delay* D(τ) = (a,b,c,d), we use a *`function FUZZY(a,b,c,d)`* to generate a delay value in *trapezoidal possibility distribution* (a,b,c,d). In *`function FUZZY(a,b,c,d),`* a random value *`atime`* in the interval [a,d] is generated first. Since the *possibility* is 1 in the interval [b,c], *`atime`* will be picked up as the delay value if *`atime`* is in the interval [b,c]. If *`atime`* is in the interval [a,b) or (c,d], we compute the *possibility* D(`atime`) in the *trapezoidal distribution* and generate a random value in (0,1). `atime` can be picked up as the delay value only if D(`atime`) ≥ the random value in (0,1). Repeat the procedure until a delay value can be picked up.

6.2 Design/CPN implementation of EFTN models for the NICE

Fig. 12 shows Design/CPN implementation of our EFTN model for a CAVE as a distributed component in the NICE. And Fig. 13 shows Design/CPN implementation for the EFTN model of garden server and communication interface between two NICE clients and the server.

We want to analyze remote avatar's display behavior on local screen and the response time from the time that a client sends out a world_changing message from its OUT key to the time that the client receives the world_changed response from the server via the IN key. So, in Fig. 12, a *timestamp* `origt` is attached to each token generated by firing transition Head_Wand_Input. The token (tracker data) will carry the *timestamp* `origt` when it is used to draw images on the local screen and it is sent to the avatar repeater and broadcasted to all other clients. A VE (e.g., CAVE) as a distributed component in NICE, will need local avatar state information, remote avatars state information, and world state information to render images. So, in Fig. 12, transition DrawComplete fires by using a token carrying local tracker data's original *timestamp* (`origt`) and remote tracker data's origin time (`origtk`). By recording `origt`, `origtk`, and the *timestamp* that transition DrawComplete fires, remote avatar's display behavior can be clearly evaluated. Section 7 will give the detail of the analysis.

Similarly, in order to analyze the response time for a client's world_changing message, we attach a *timestamp* `sendt` to each token sent to the client's OUT key (e.g., places 1Out, 2Out) when the transition SendIt fires. And we record the input token's *timestamp* `sendt`, transition ReceiveWorldEvent's firing time, and the ID of the world_changing event's initiator (world_changing avatar's ID), when transition ReceiveWorldEvent fires. As we'll see in Section

7, the characteristic and bottleneck of the response time via the TCP channel can been easily captured.

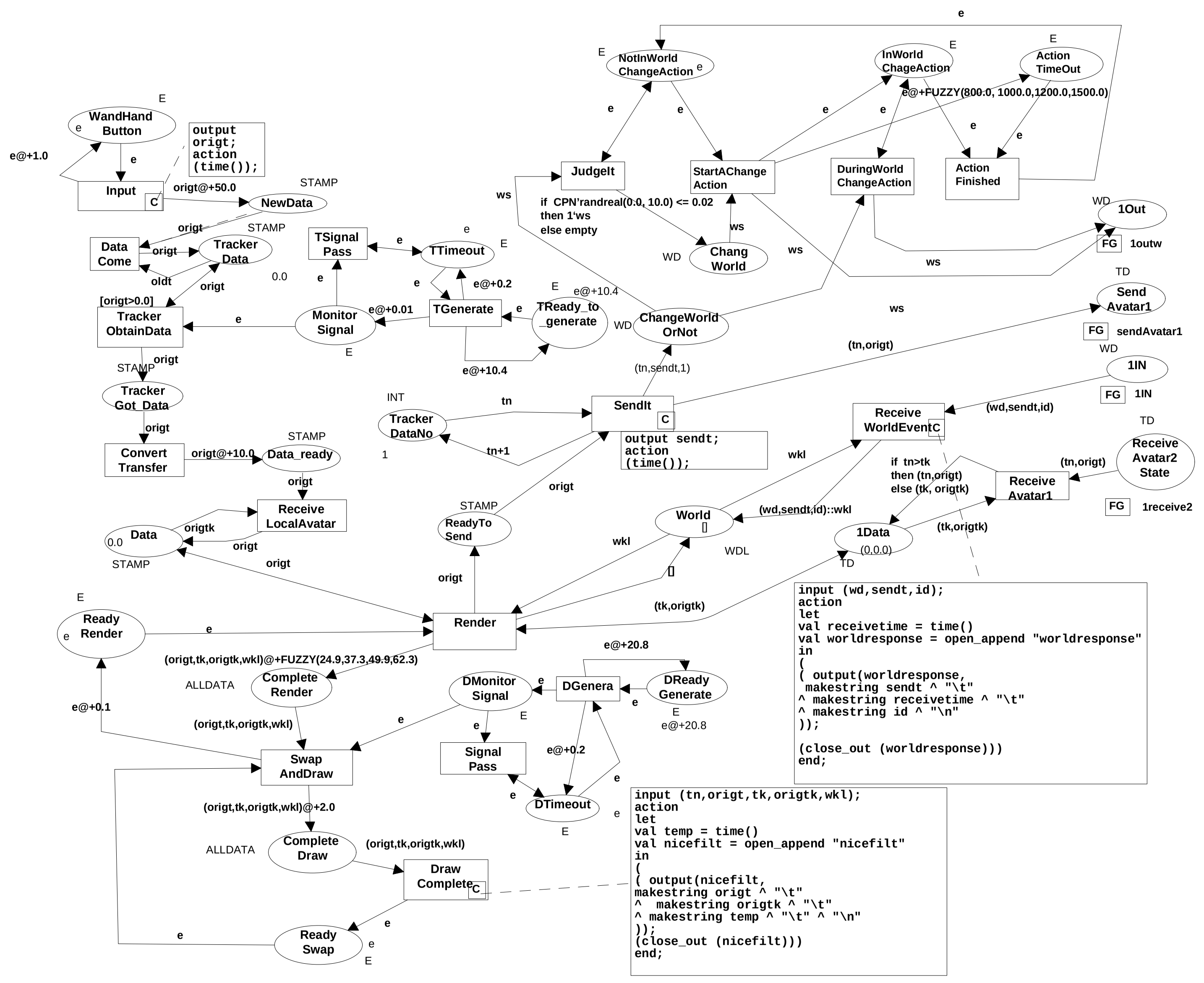
6.3 Design/CPN implementation of an EFTN model for TCP Protocol

The Transmission Control Protocol (TCP) is a reliable and ordered transport layer protocol. The data transmission in TCP is basically a sliding window mechanism, where the window size is advertised by the receiver. Fig. 14 shows a Design/CPN implementation of an EFTN model for TCP protocol. The features of TCP modeled in Fig. 14 are as follows:

- ♦ Window size advertised by the receiver: assuming that the receiver's window size is 64K initially and that each data unit is 1024 byte, for simplicity, we initialize receiver's window size as 64. An ACK message sent by the receiver is composed of a sequence number, the sequence number of next data unit that the receiver is expecting, and receiver's available window size. The receiver's available window size tells the sender how many bytes the receiver can accept.
- ♦ Congestion window: the sender uses 2 windows to determine how many bytes it can send. One is the receiver advertised window, and the other is the congestion window that the sender uses to detect network congestion. If the number of bytes stored in the sender's buffer waiting for ACK ≤ *min* (*receiver_advertised_window, congestion_window*), the sender can keep on transmitting new coming data. Otherwise, no new data can be transmitted.
- ♦ Slow start and Congest Avoidance: the sender's congestion window is initialized as 1 data unit. The sender has a parameter: *threshold*, initialized as 64. Once a data unit in the sender buffer is time out before the sender receives an ACK message for it, the sender sets: *congestion_window* =1 and *threshold* = *max* (1, 0.5*(*min*(*congestion_window, receiver_window*))). Once a data unit in the sender buffer has been acknowledged, the sender will grow the congestion window. At that time, if *congestion_window* ≤

threshold, then the sender is in the slow start mode and *congestion_window* = *congestion_window* + 1; if *congestion_window* > *threshold*, then the sender is in the congest avoidance mode and *congestion_window* = *congestion_window* + (1 / *congestion_window*).

- Delayed ACK: the receiver will not send an ACK right after a data unit is received. An ACK corresponding to a data unit i will wait for 200 ms before being sent. However, if the ACK for data unit i+1 comes when ACK for i is waiting, the receiver will cancel ACK for i and send ACK for i+1 immediately. Also, when an out-of-order data unit arrives, the receiver will discard the data unit and send an ACK immediately to tell the sender which data unit the receiver is expecting.
- Retransmission timer and persistence timer: when a data unit is sent, a retransmission timer is started. If the retransmission timer expires before the data unit is acknowledged, the sender will resend that data unit. Most TCP implementations use *retransmission timeout* = *RTT + 4 * D*, where *RTT* is the best estimate of round-trip time and *D* is the estimation of standard deviation. The retransmission timer should be dynamically updated. But for simplicity, we set it to 550 ms in our model. The persistence timer is used to prevent the following deadlock: the receiver advertised the receiver window as 0, so the sender will stop transmitting and wait for the receiver to update the receiver window. Later, the receiver advertises a larger available window size. However, this ACK message is lost on the way. Then the sender and receiver will wait for each other. To prevent this deadlock, the persistent timer will be starter once the receiver advertised the window as 0. If the persistent timer expires and the receiver window is still 0, the sender will send the receiver a probe. Once the probe reaches the receiver, the receiver will immediately send an ACK with the current receiver-window size.



```
output
origt;
action
(time());
```

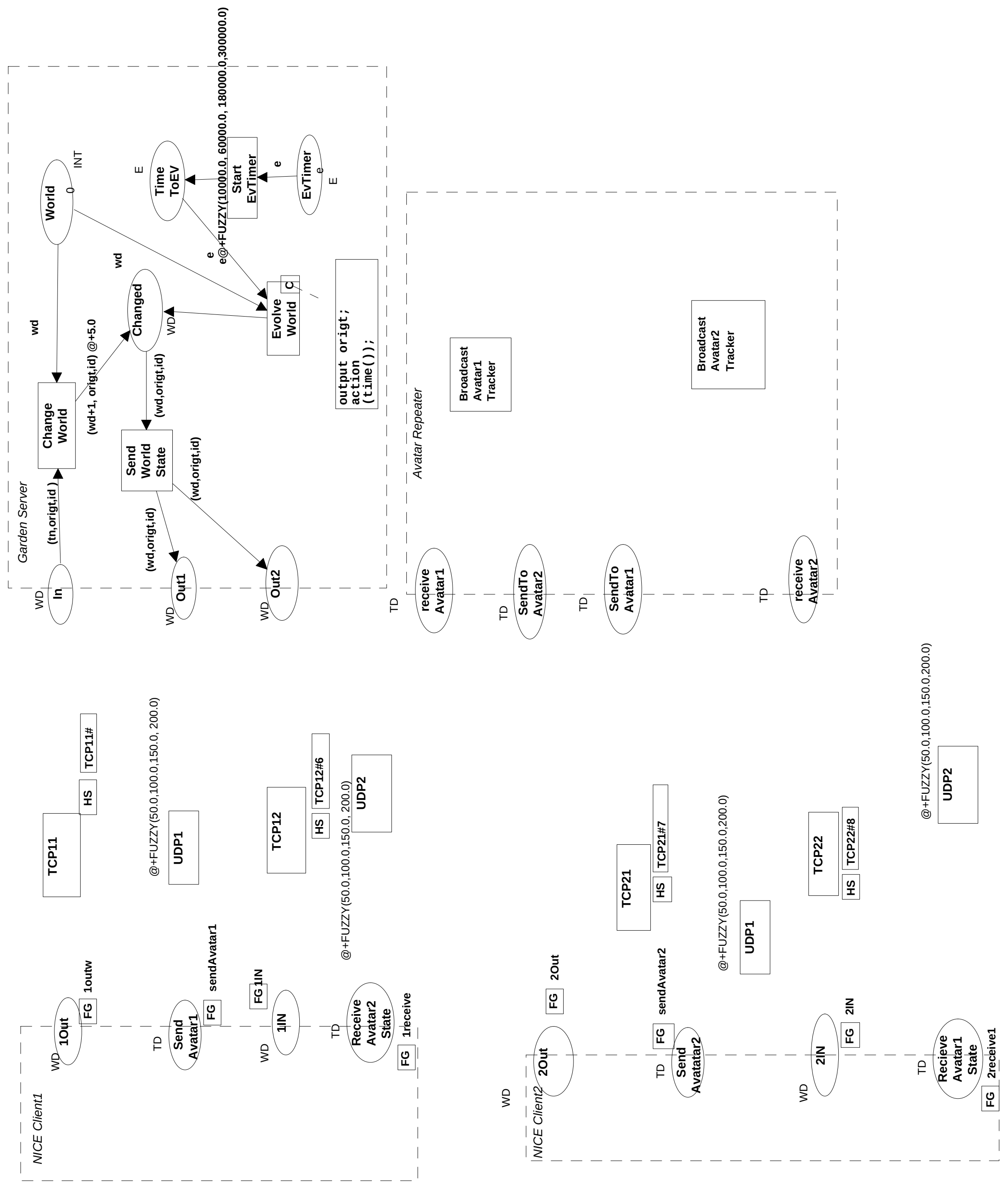
```
output sendt;
action
(time());
```

```
input (wd,sendt,id);
action
let
val receivetime = time()
val worldresponse = open_append "worldresponse"
in
(
( output(worldresponse,
 makestring sendt ^ "\t"
^ makestring receivetime ^ "\t"
^ makestring id ^ "\n"
));

(close_out (worldresponse)))
end;
```

```
input (tn,origt,tk,origtk,wkl);
action
let
val temp = time()
val nicefilt = open_append "nicefilt"
in
(
( output(nicefilt,
makestring origt ^ "\t"
^  makestring origtk ^ "\t"
^ makestring temp ^ "\t" ^ "\n"
));
(close_out (nicefilt)))
end;
```

Fig. 12 Design/CPN implementation of EFTN model for a Nice client


Garden Server
World
0
INT
Time ToEV
E
Start EvTimer
e@+FUZZY(10000.0, 60000.0, 180000.0,300000.0)
e
EvTimer
e
E
wd
Changed
WD
Evolve World
C
output origt;
action
(time());
wd
Change World
(wd+1, origt,id) @+5.0
(wd,origt,id)
Send World State
(wd,origt,id)
(tn,origt,id)
(wd,origt,id)
In
WD
Out1
WD
Out2
WD
Avatar Repeater
Broadcast Avatar1 Tracker
Broadcast Avatar2 Tracker
receive Avatar1
TD
SendTo Avatar2
TD
SendTo Avatar1
TD
receive Avatar2
TD
TCP11
HS
TCP11#
@+FUZZY(50.0,100.0,150.0, 200.0)
UDP1
TCP12
HS
TCP12#6
@+FUZZY(50.0,100.0,150.0, 200.0)
UDP2
NICE Client1
1Out
WD
FG
1outw
Send Avatar1
TD
FG
sendAvatar1
1IN
WD
FG
1IN
Receive Avatar2 State
TD
FG
1receive
TCP21
HS
TCP21#7
@+FUZZY(50.0,100.0,150.0,200.0)
UDP1
TCP22
HS
TCP22#8
@+FUZZY(50.0,100.0,150.0,200.0)
UDP2
NICE Client2
2Out
WD
FG
2Out
Send Avatatar2
TD
FG
sendAvatar2
2IN
WD
FG
2IN
Recieve Avatar1 State
TD
FG
2receive1

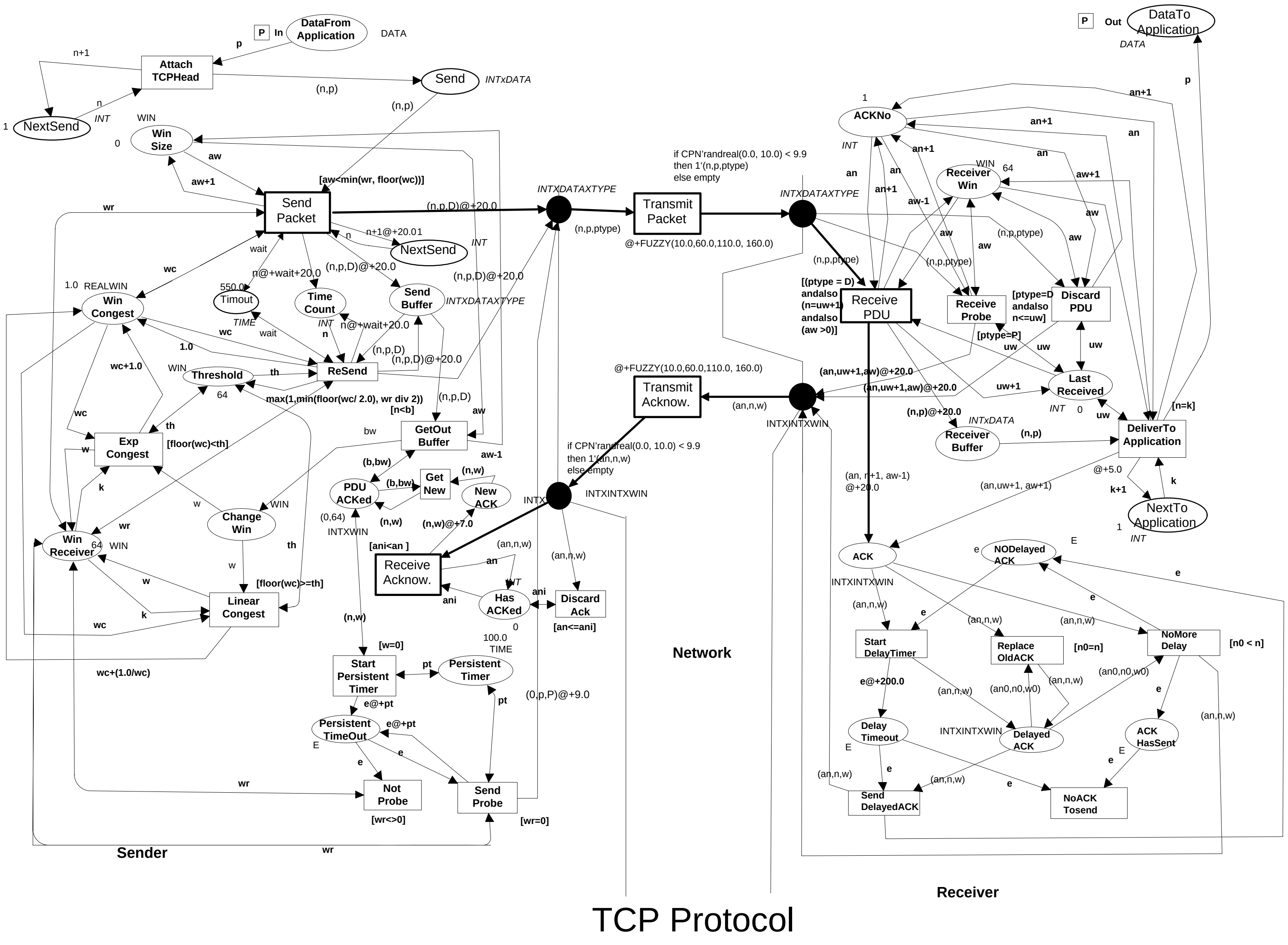


Fig. 14 Design/CPN implementation of EFTN model for TCP protocol

```
color INT = int timed;
color WIN = int;
color STAMP = real timed;

color AVATARID = INT;
color WD = product INT * STAMP * AVATARID;
color WDL = list WD ;
color DATA = WD ;

color TYPE = with D | P ;
color INTxDATA = product INT * DATA;
color INTXDATAXTYPE = product INT * DATA * TYPE;
color INTXINTXWIN = product INT * INT * WIN;
color INTXWIN = product INT * WIN;
color E = with e timed;
color TD = product INT * STAMP;
color ALLDATA = product STAMP * INT * STAMP * WDL;
color REALWIN = real timed;

var n, n0 : INT;
var p,str : DATA;
var aw, w,w0,k, wr, th, bw : WIN;
var wc : REALWIN;
var b, uw, an,ani, an0 : INT;
var ptype: TYPE;
var wkl : WDL;
var wait, pt : TIME;
var id : AVATARID;
var ad : ALLDATA;
var td : TD;
var origt, origtk, oldt, origw, sendt : STAMP;
var tn, tk : INT;
var wd : INT;
var ws, wk : WD;

fun FUZZY(a: real,b: real,c: real,d: real) =
let
val atime = CPN’randreal(a,d);
in
(if atime >= b andalso atime <= c
then atime
else if (atime <b)
 then  if (a<b)
  then if ((atime-a)/(b-a) >= CPN’randreal(0.0, 1.0)) then atime
      else FUZZY(a,b,c,d)
  else if (atime = a) then a
    else  FUZZY(a,b,c,d)
 else
  if (d>c)
  then  if ((atime-c)/(c-d)+1.0 >=
CPN’randreal(0.0, 1.0)) then atime
      else FUZZY(a,b,c,d)
  else if (atime = c) then c
   else FUZZY(a,b,c,d)
)
end;
```

Fig. 15 Global Declaration node in Design/CPN implementation

7 Simulation Results

7.1 Remote Avatar's Display Behavior on Local Screen

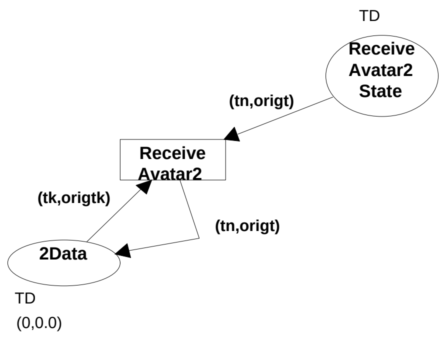


Fig. 16 The remote avatar data receiving without using the filter

Unreliable protocols (e.g. UDPs) are used for the transmission of avatar state information (remote tracker data). That is because: 1) The loss of one tracker data is usually followed shortly afterwards by newer ones; and 2) Unreliable protocols have a lower latency and utilize lower bandwidth than reliable protocols. However, UDP protocol is unordered. Using unordered remote tracker data will make the remote avatar jump back and forth on the local screen. Originally, a NICE client uses remote avatar tracker data in their arriving order. Fig. 16 shows how the transition ReceiveAvatar2 works in the original design. Fig. 17 shows the time of avatar2's original movement and the time that movement is displayed on NICE client1's screen in the original design. We can see that the remote avatar's display may jump back and forth. To avoid the jumping back behavior, the NICE currently uses a filter to accept remote avatar data in increasing order. As shown in Fig. 12, the transition ReceiveAvatar2 works as a filter. Fig. 18 shows the display behavior using the filter. Now the jumping back phenomenon is eliminated. However, one early arriving remote avatar tracker data will make the filter discard all remote tracker data that are sent before, but received later than that early arriving data. From Fig. 18, we can see that the display of the remote avatar is not very smooth. Fig. 19 shows the distribution of the delay from the time that a remote avatar has a movement to the time that remote tracker information is displayed on local screens. And Fig. 20 shows the distribution of the time that remote tracker data lags behind the local data.

To display the remote avatar's movement more smoothly, we can use a buffer to store the incoming remote avatar state information sent after the last one used for local display. And we use the remote avatar's state information in smooth gap. Fig. 21 shows the improved strategy using a buffer for remote avatar tracker data. Once a new remote tracker data comes, if its sequence number is greater than the last one's used for rendering images, the `function inserttd` (as shown in Fig. 22) inserts the new data into the buffer and keeps the list of sequence numbers of the remote tracker data in ascending order in the buffer. When it is time to render new images, we pick up the remote tracker data in the middle of the list from the buffer. Fig. 23 shows that the display of the remote avatar is much smoother using the improved strategy than using the filter.

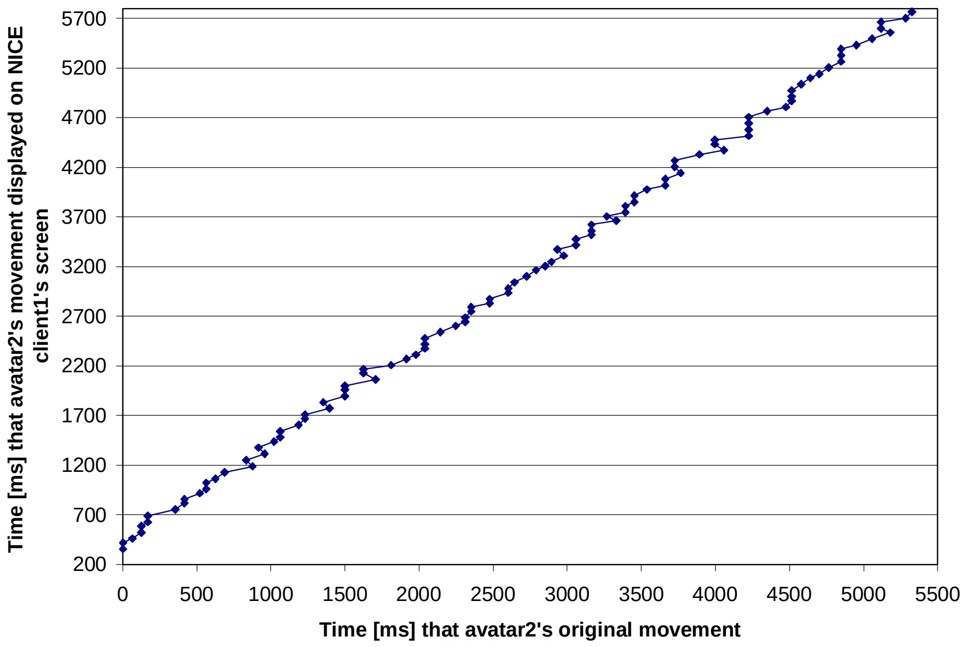


Fig. 17 The simulated display behavior of a remote avatar on the local screen without using the filter

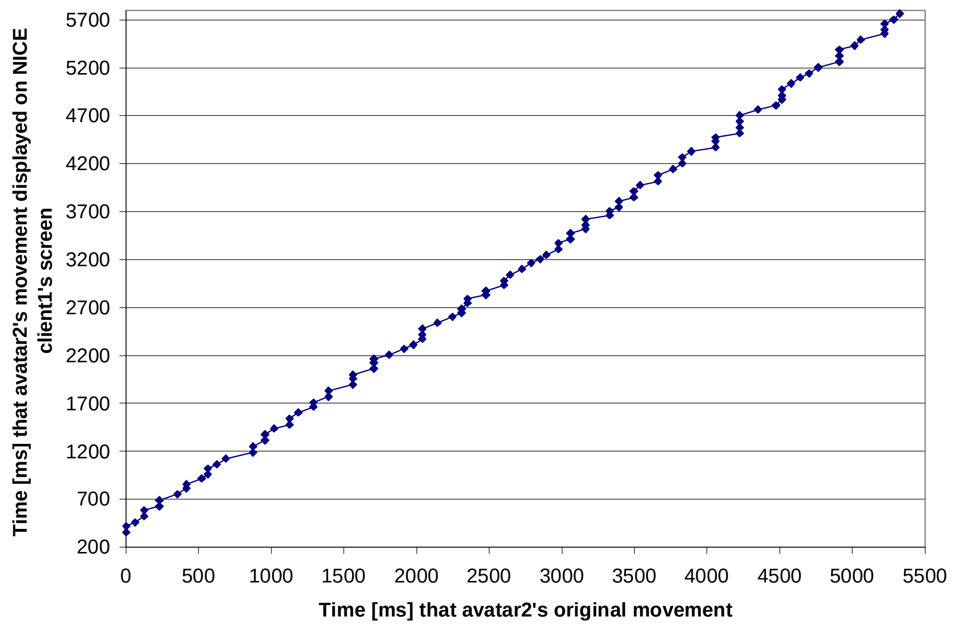


Fig. 18 The simulated display behavior of a remote avatar on the local screen with using the filter

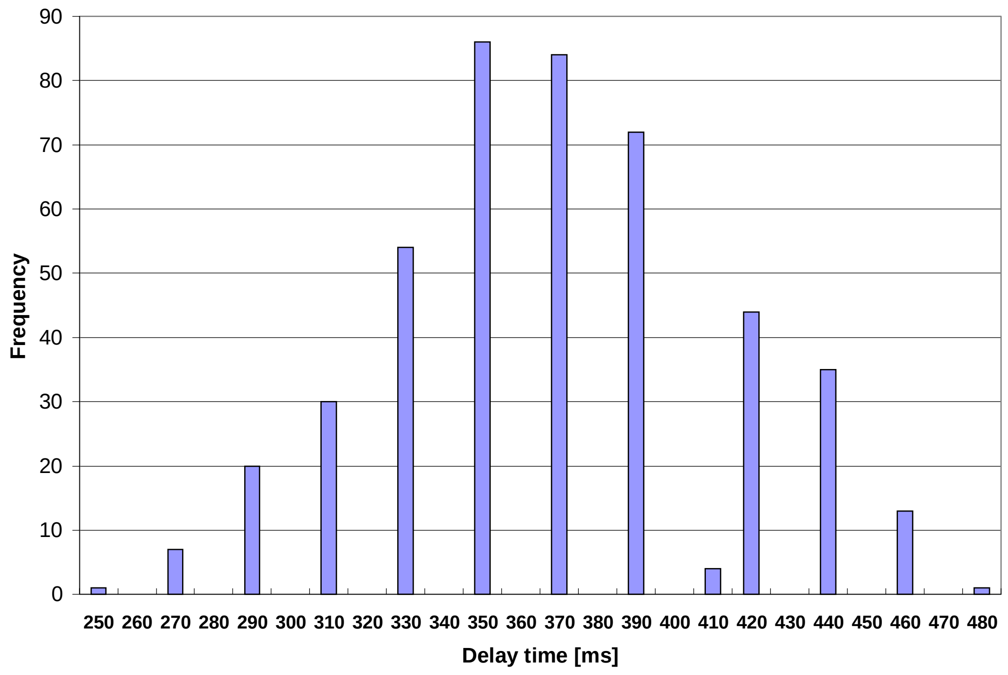


Fig. 19 The distribution of the delay from the time that a remote user has a movement to the time that movement is displayed on the local screen in simulation.

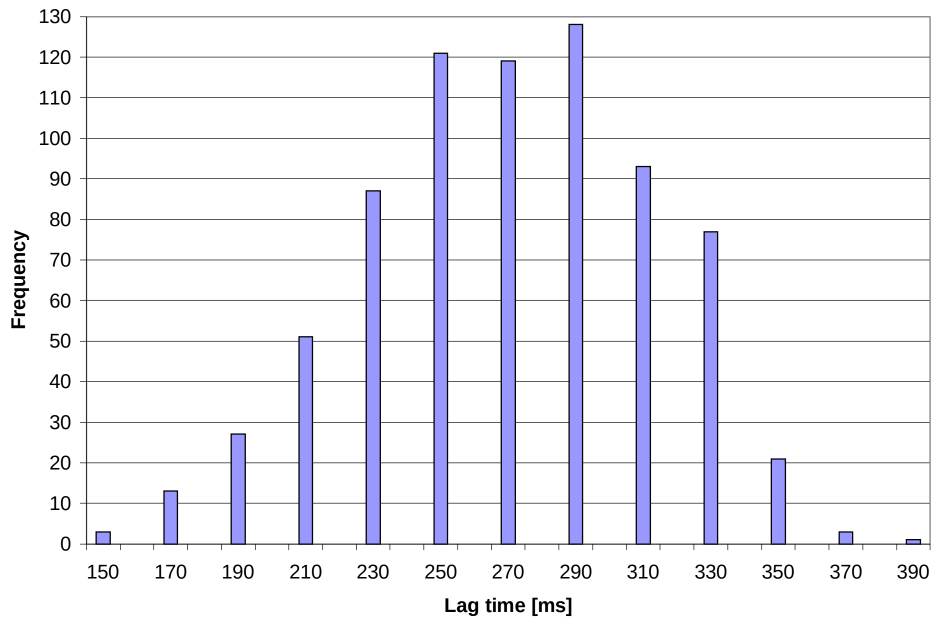
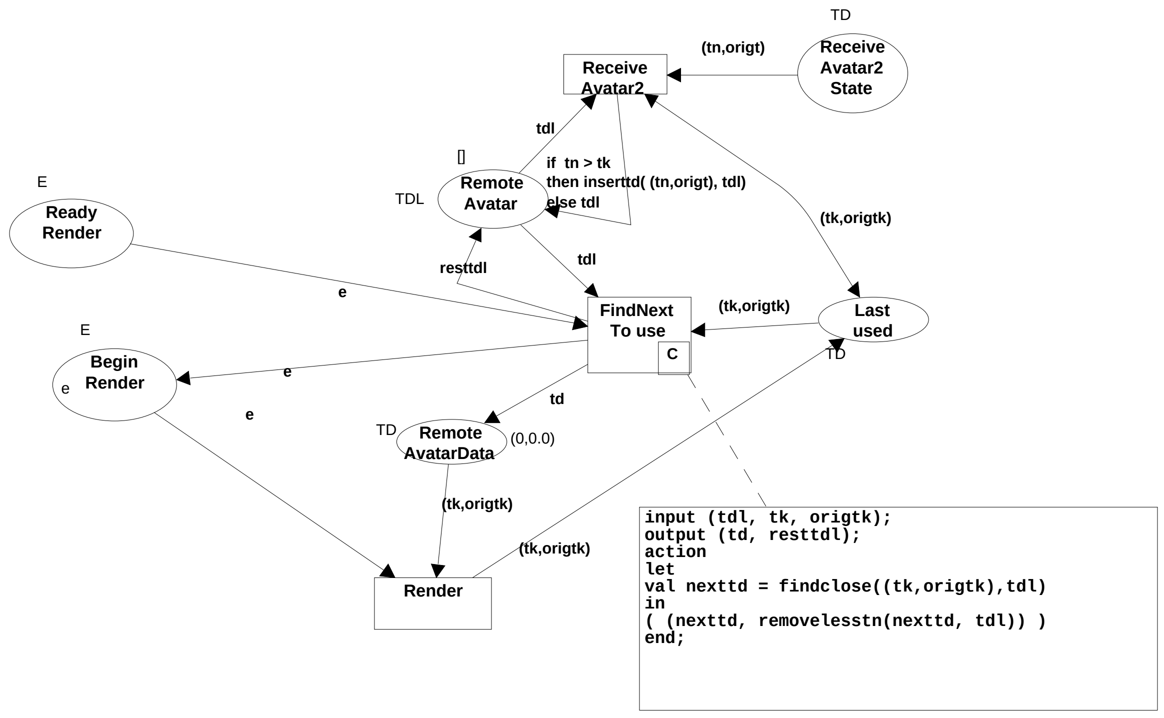


Fig. 20 The distribution of the time that remote avatar's display lags behind local avatar in simulation



```
input (tdl, tk, origtk);
output (td, resttdl);
action
let
val nexttd = findclose((tk,origtk),tdl)
in
( (nexttd, removelesstn(nexttd, tdl)) )
end;
```

Fig. 21 Improved strategy: use a buffer for remote avatar state information

```
color TD = product INT * STAMP;
color TDL = list TD ;

var td, td1 : TD;
var tdl, resttdl, tdl1 : TDL;



fun mid(number: int) = if number mod 2 = 0 then number div 2
      else number div 2 +1 ;

fun greater((tn,_): TD, (tk,_): TD) = if tn > tk then true else false;

fun muchgreater((tn,_): TD, (tk,_): TD) = if tn > (tk+2) then true else false;


fun findclose(td, tdl) =
let
val numbertd = length(tdl)
in
( if numbertd = 0 then td
  else if (numbertd = 1) andalso muchgreater(hd(tdl),td) then td
  else  nth(tdl,  mid(numbertd) - 1)
)
end;

fun removelesstn(_, nil) = nil
| removelesstn(td,  td1 :: tdl1) = if greater(td,td1)  then removelesstn(td, tdl1)
     else tdl1 ;

fun inserttd(td, nil)  = [td]
|   inserttd(td, td1 ::tdl1) = if greater(td1,td) then td::td1::tdl1
        else td1::inserttd(td, tdl1);
```

Fig. 22   Definition of functions used in improved strategy for remote avatar state information

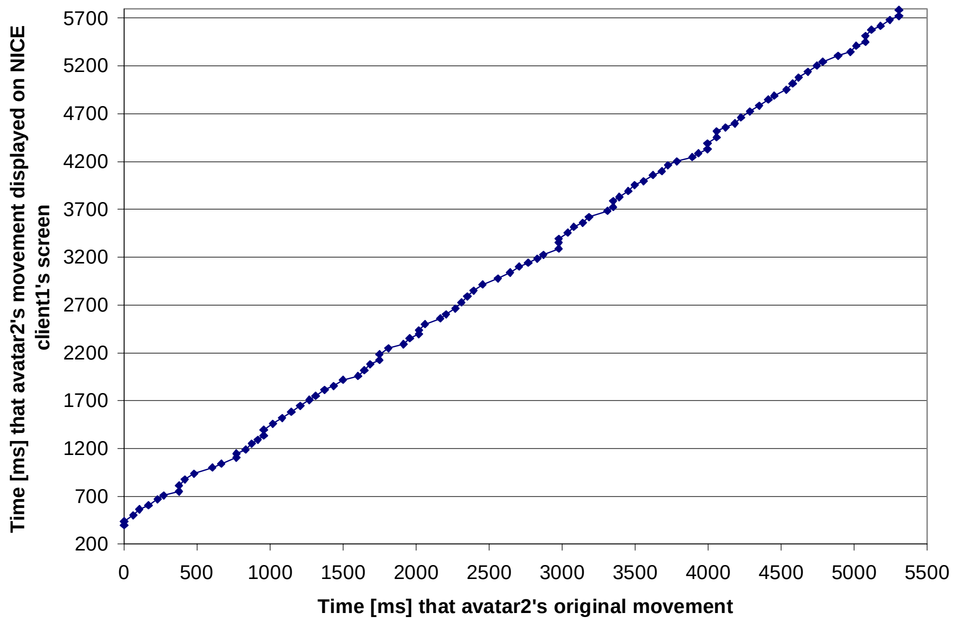


Fig. 23   The simulated display behavior of a remote avatar on the local screen with using buffer

7.2 *Test of TCP protocol*

We tested our model for TCP protocol by giving a data unit to the TCP sender for transmission every 50 ms, and every 100 ms, respectively, and recording the delay from the time that the data unit is passed to the TCP sender to the time that the TCP receiver delivers the data unit to the application. Our simulation result is consistent with the experimental results in [9], which is obtained by monitoring the network delay on internet.

- *Case 1*: give the TCP sender a data unit every 50 ms:

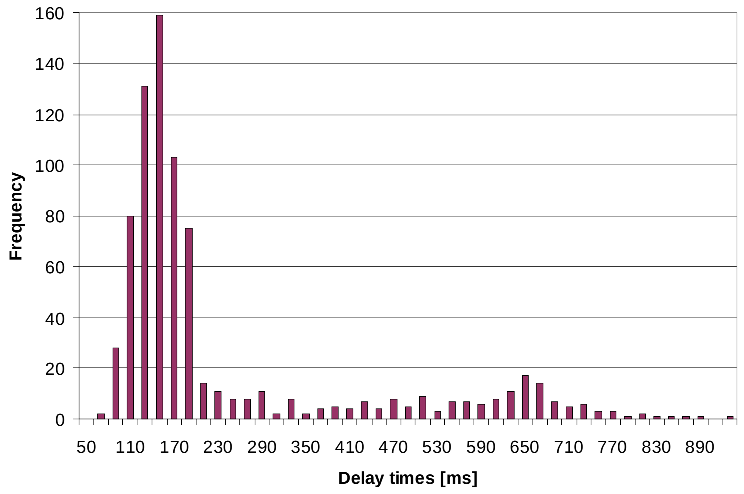


Fig. 24(a) Delay distribution for data unit coming every 50 ms

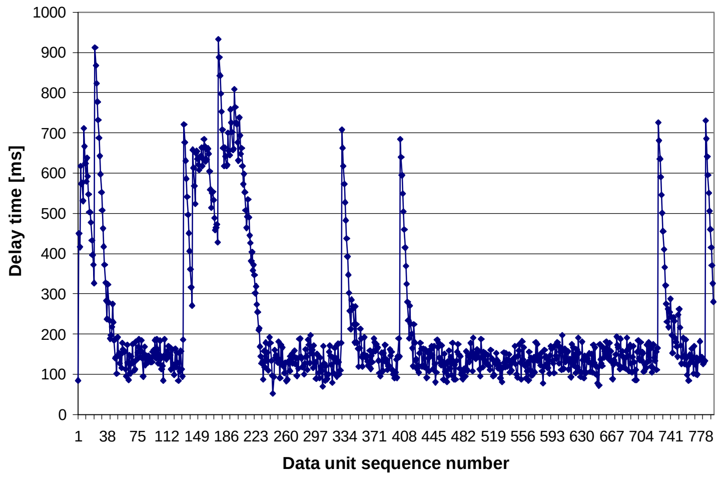


Fig. 24 (b) History chart for each data unit (for data unit coming every 50 ms)

As shown in Fig. 24 , it is obvious that the TCP has a slow start and one data unit's long delay will delay all subsequent data units after it starts. TCP's reliable and ordered behavior greatly increases the average network latency and jitter.

♦ *Case 2*: give the TCP sender a data unit every 100 ms:

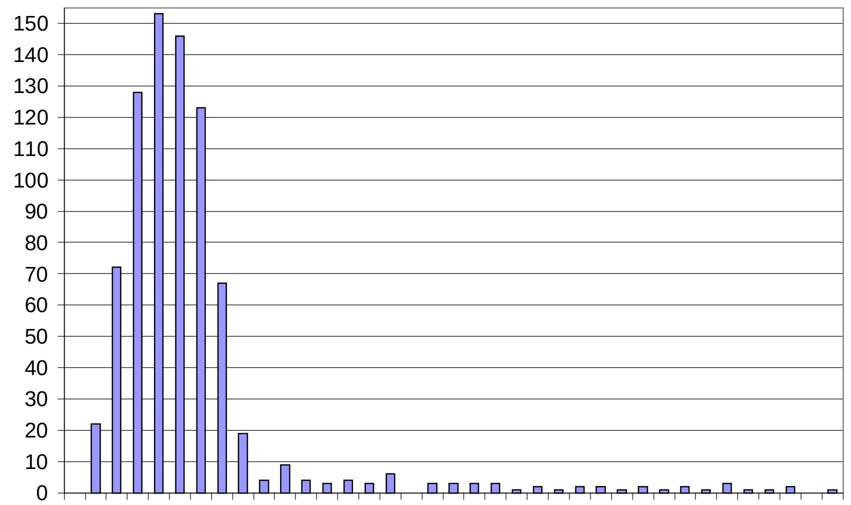


Fig. 25 (a) shows that decreasing the traffic to 1 data unit every 100 ms makes the delay distribution very similar to UDP's delay distribution (50,100,150, 200) ms. However, Fig. 25 (b) shows that 1 data unit's long delay (because of loss and retransmit), still greatly delays the subsequent data units. Also the slow start still exists.

7.3 Response Time via TCP channel

By using the method described in Section 6, we can analyze the response time for a client's world_changing message.

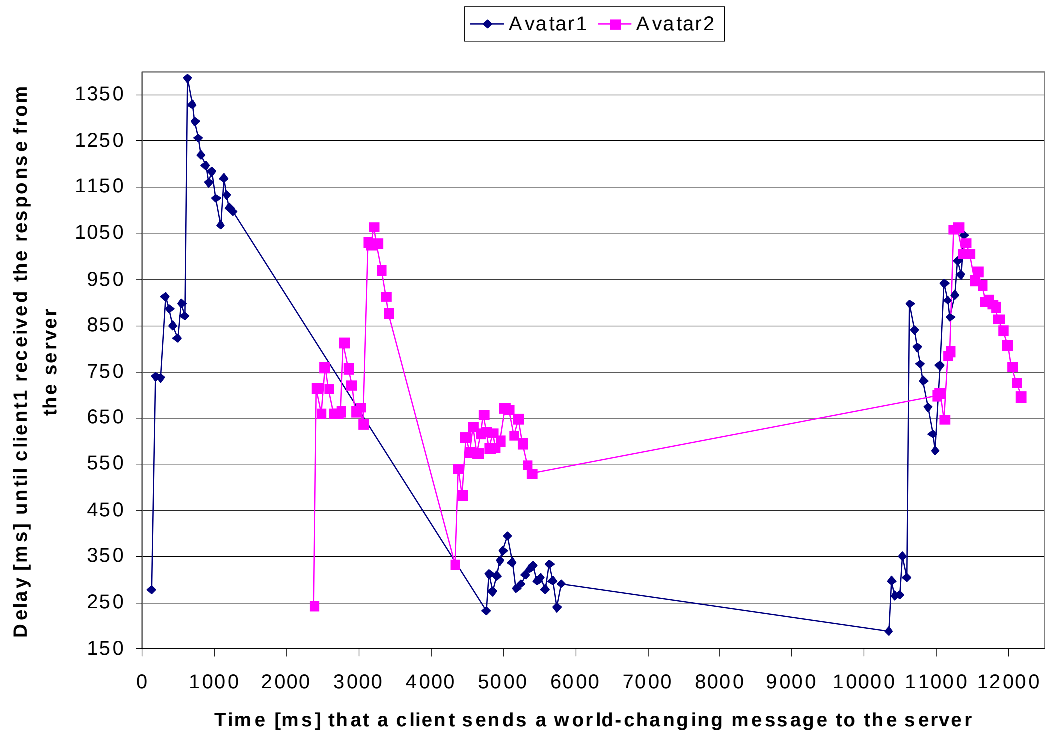


Fig. 26 History chart of the response time (from time that a message is sent out by client1 (or client2) to time that client1 get the broadcasted response from server via TCP channel) in simulation.

Fig. 26 shows the history chart recorded at client1's site for the NICE with 2 clients. Avatar1 has 3 world-changing actions starting around 125, 4763, and 10338 ms, respectively. Avatar2 has 3 world-changing actions starting around 2371, 4326, and 11003 ms, respectively. Each action takes around 1000 ms. We can see the effects of TCP's slow start. Also, avatar1 and avatar2 tried twice to change the world at the same time, one after 4763 ms, the other after 11003 ms. For the first time (after 4763 ms), a long delay happens on the TCP channel from client2's OUT key to the server's IN key. If two data are sent from client1 and client2 to the server at the same time and client2's data arrives much later than client1's, each one of client2' subsequent data arrives at the server later than client1's corresponding data (the data sent from client1 at almost the same time as client2). Then the response for each of cleint2's data will be behind the response for client1's corresponding data, in the queue of responses from the server to client1. So the response for each of client2's data, arrives at client1's IN key later than the response for client1's corresponding data.

For the second time (after 11003 ms), long delays happen on the TCP channel from the server's OUT key to client1's IN key after 11200 ms. The response for both of client1 and client2's data are delayed.

To create more traffic on TCP channels, we run the simulation for 5 clients. Fig. 27 shows the history chart recorded at client1's site for the NICE with 5 clients. More obviously than in the case of 2 clients, the response times for all avatars' world-changing activities have the same trend at client1's site, if it happens that all avatars want to change the world at the same moment (e.g., arround 78000 ms in Fig. 27.) A data lost on the TCP channel from the server's OUT key to a client's IN key will not only cause long delay for client1 receiving response for one avatar's world-changing activity, but also postpone client1 receiving response for all avatar' world-changing activities. The TCP channel from the server's OUT key to a client's IN key may get congested when the number of clients is increased and all avatars happen to try changing the world at the same time. In this situation, it becomes a bottleneck.

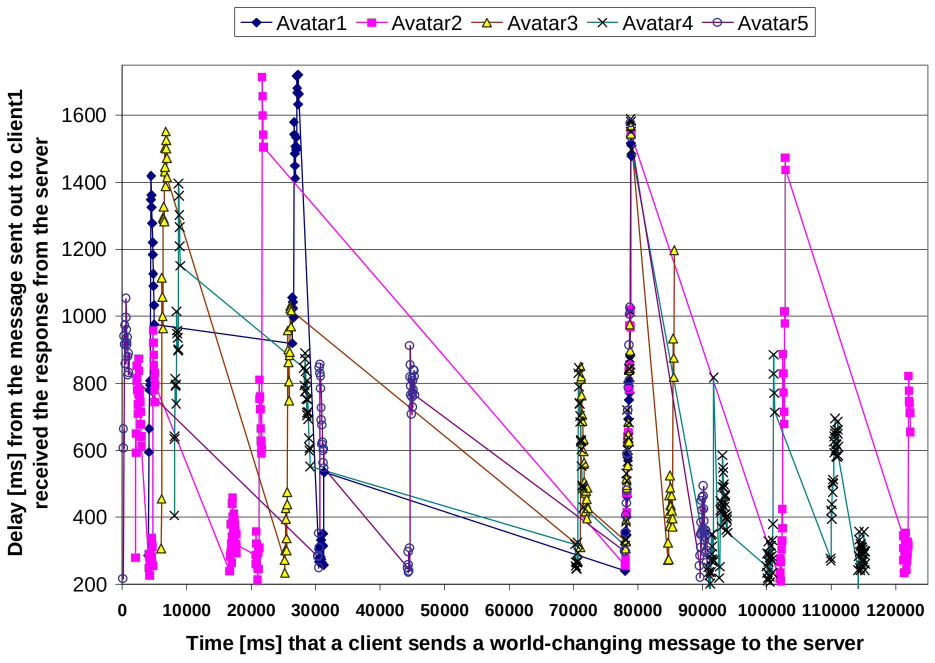


Fig. 27 History chart of the response time for the NICE with 5 clients

8 Conclusion and Future Work

CVEs demand high requirements on network delays and jitters, so that remotely distributed users' collaboration will not be disturbed. By using reduction, simulation, or occurrence graph on our EFTN model, the network effects on CVEs can be easily evaluated.

Our EFTN models for the CAVE and NICE and the analysis of our EFTN models for the NICE, indicate that EFTNs are powerful to specify and verify VRs. EFTNs can capture the temporal uncertainties in CVEs. By simulating our EFTN models, we can analyze the network effects on CVEs and the dynamic performance of CVEs.

In Section 4, we show a simple example of possibility analysis of our EFTN model for the CAVE. EFTN models can give information on partial ordered events in terms of their degrees of possibilities. The possibility analysis is based on model checking on transition firing sequences [11] or occurrence graphs. The possibility analysis of our EFTN models for CVEs is to be included in our future paper.

The simulation indicates that TCP's reliable and ordered behavior greatly increases the average network latency and jitter. Thus, it is desirable to design a new transport layer protocol which is suitable for transmitting world state information with shorter latency and lower jitter than TCP. We plan to propose new protocols, model and analyze theirs performance and effects on CVEs in our future paper.

ACKNOWLEDGEMENTS

We sincerely thank Robert Kenyon for explaining CVEs, Jason Leigh and David Pape for clarifying CVE implementation details, Andrew Johnson and Dan Sandin for discussing problems that exist in CVEs, and all members of the Electronic Visualization Laboratory for providing valuable comments on our work.

References

[1] E. Juan, J. P. Tsai, T. Murata, and Y. Zhou, "Reduction Methods for Real-Time Systems Using Delay Time Petri Nets," Technical report, EECS Dept., University of Illinois, Chicago, March, 1999.

[2] M. Roussos, A. Johnson, T. Moher, J. Leigh, C. Vasilakis, C. Barnes, "Learning and Building Together in an Immersive Virtual World," To appear in Presence vol. 8, no. 3, June, 1999.

[3] A. Johnson, M. Roussos, J. Leigh, C. Barnes, C. Vasilakis, T. Moher, "The NICE Project: Learning Together in a Virtual World," in the proceedings of VRAIS '98, Atlanta, Georgia, Mar 14-18, 1998, Pp 176-183.

[4] D. Pape, "CAVE user's guide," Electronic Visualization Laboratory, University of Illinois at Chicago, Dec. 1996.

[5] P. Merlin, "A study of the Recoverability of Computer Systems," Ph.D thesis, Computer Science Dept., University of California, Irvine, 1974.

[6] R. Mascarenhas, D. Karumuri, U. Buy, and R. Kenyon, " Modeling and analysis of a virtual reality system with time Petri nets," Procs. 19th Int. Conf. on Software Engineering, pp. 33-42, April 1998, Kyoto, Japan.

[7] T. Murata, "Temporal Uncertainty and Fuzzy-Timing High-Level Petri Nets," Invited paper at the 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan,, LNCS Vol. 1091, pp. 11-28, Springer-Verlag, New York, June 1996.

[8] T. Murata, "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, Vol. 77, No 4, April, 1989, pp. 541-580.

[9] K. Park, and R. Kenyon, “Effects of Network Characteristics on Human Performance in a Collaborative Virtual Environment,” Proceedings of IEEE VR `99 , Houston TX, March 13-17, 1999.

[10] T. Murata, T. Suzuki and S. Shatz, "Fuzzy-Timing High-Level Petri Nets (FTHNs) for Time-Critical Systems," in J. Cardoso and H. Camargo (editors) "Fuzziness in Petri Nets" Vol. 22 in the series "Studies in Fuzziness and Soft Computing" by Springer-Verlag, New York, pp. 88-114, 1999.

[11] Y. Zhou and T. Murata, “Petri Net Model with Fuzzy-Timing and Fuzzy-Metric Temporal Logic,” to appear in the special issue on fuzzy Petri nets: concepts and intelligent system modeling, International Journal of Intelligent Systems, 1998.

[12] Y. Zhou and T. Murata, “Fuzzy-Timing Petri Net Model for Distributed Multimedia Synchronization,” in the Procs. of the 1998 IEEE International Conference on Systems, Man, and Cybernetics (SMC’98), La Jolla, Calif., Oct. 12-14, 1998.

[13] Y. Zhou, T. Murata, and J. Tsai, “Reduction Methods for Real-Time Systems Using Fuzzy Timing Petri Nets,” Technical report, EECS Dept., University of Illinois, Chicago, 1999.

[14] K. Jensen, and Design/CPN group, “Design/CPN Online,” Department of Computer Science, University of Aarhus, Denmark . Online: http://www.daimi.au.dk/designCPN/.

[15] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, “Virtual Reality: The Design and Implementation of the CAVE,” in Proceedings of SIGGRAPH '93 Computer Graphics Conference, ACM SIGGRAPH, August 1993, pp. 135-142.

[16] T. A. DeFanti, D. J. Sandin, and C. Cruz-Neira, “A `Room' with a `View',” IEEE Spectrum, October 1993, pp. 30-33.

Table of Contents

Table of Figures