Distributed Systems

Distributed Simulation of Timed Petri Nets

A Modular Approach Using Actors and Time Warp

This modular and actor-based approach to the development of time-dependent distributed systems separates functional aspects from timing. The Time-Warp mechanism regulates the interaction policy among logical processes, shifting overhead from communication to computation.

his work focuses on using a modular and actor-based framework to develop distributed and time-dependent systems—both real-time and simulated. Our approach is modular in that it specifies and handles functional and timing requirements separately. For example, application actors remain unaware of timing constraints—that is, when and why messages activate them. They provide functional responses to messages belonging to a given message interface. Timing issues fall to reflective actors that transparently filter message transmissions within groups of actors, applying timing classes that ultimately affect scheduling. Our approach favors object reusability and proves useful both for developing hard real-time systems and for supporting general distributed discrete-event simulation.

In this article, we specialize our actor model to the distributed simulation of timed Petri nets, or TPNs, using Time Warp. The resultant architecture supports different timing models and enabling and firing strategies. Our Time-Warp mechanism minimizes the communication overhead during rollbacks, and we can tune it to model partitioning and load balancing for good performance in a heterogeneous, distributed context.

We also describe our development of a test bed for our simulation framework, a TPN model for complex cellular networks using a dedicated and homogeneous distributed system of Risc6000 workstations connected by a Fast Ethernet LAN. Our simulation of such networks exhibits good performance—for example, a speedup factor of 8 with 10 processors.

An actor model for time-dependent systems

The variant of the Actor model that we use—DART—was designed for the development of distributed and time-dependent systems. By this model, a system consists of a collection of subsystems, one per processor, interconnected through a possibly deterministic communications system. Each subsystem hosts a control machine, a scheduler, and a set of application actors. The discipline of programming in-the-large often suggests that only one administrator...
actor should be referenced at the system level. This administrator actor receives
message requests and can delegate its processing to inner subsystem actors. In a
sense, the administrator represents the subsystem. The control machine provides
the necessary support for scheduling, message selection, and message dispatch.

ACTORS
Actors are the basic building blocks in the small. We can think of an actor as a
finite-state machine that evolves through a life cycle—that is, a succession of states
or behaviors. An actor is a reactive object, responding to incoming messages on the
basis of its current state and message content. Message reception is implicit—that
is, there is no receive primitive. An actor is at rest until a message arrives. Message
processing triggers a state transition and the execution of an action. Because action
execution is atomic, it cannot be preempted or suspended. Messages can be unexpected
at an actor in its current state. An actor can postpone processing such messages by remembering them in local
data or in states of the life cycle.

Basic operations on DART actors are:

- new, which creates a new actor,
- become, which changes the actor's state,
- send, by which an actor transmits messages to acquaintance actors (including itself) in a nonblocking way.

DART differs from the standard Actor model in several ways:

- Its actors have no internal thread. DART actors are instances of a passive class that directly or indirectly extends the Actor base class. The actors' life cycles come from a message handler that the control machine invokes at message dispatch time.
- It does not include a mail queue per actor. Rather, the control machine introduces a single message queue on which it schedules, selects, and dispatches sent messages according to a proper control structure.
- The source of concurrency among actors in a single subsystem is action inter-
leaving ensured by the control-machine dispatching scheme. Concurrency among actors allocated to different processors is true parallelism.

Besides actors, the architecture also allows for the existence of passive objects that don't have a life cycle. The system invokes their methods synchronously by the usual procedure-call semantics. Because of the atomic character of action execution in actors, passive objects behave naturally as monitors, so there is no need for conventional mutual-exclusion primitives (semaphores and derivatives). The system could also include an actor to export accessor methods that can be requested synchronously instead of by message passing.

REFLECTION AND CONTROL

The actors' dynamic behavior depends on a control discipline adopted at the control-machine level. Here, the control structure can be purely event-driven as in the actor model, which is useful for concurrent applications; it can be purely time-driven to suit hard real-time systems; or it can be a combination of the two. The control machine relies on the concept of reflection. It can reason in a time dimension and be causally connected with application actors. As shown in Figure 1, the basic components of a control machine are:

- the clock,
- the message queue (a priority queue used as a calendar or event list),
- the controller, and

- the programmer-defined scheduler actor.

At initialization, the control machine is fed a particular scheduler. The time notion can be virtual or real. In a simple case, each message comes along with a time stamp indicating the occurrence time of the event captured by the message. In general, for real-time applications, a message comes accompanied by a validity time window—[min, max]—that specifies possible delivery times.2,3

Let's focus here on virtual time only and a stand-alone simulation control machine. This system attaches a time stamp to each message. The controller method is equipped with the maximum

---

**Acronyms and abbreviations**

- **DART** Distributed Architecture for Real Time
- **FIFO** first-in, first-out
- **GSPN** generalized stochastic Petri nets
- **GVT** global virtual time
- **IQ** input queue
- **LP** logical process
- **LVT** local virtual time
- **MAC** modified aggressive cancellation
- **MNSV** maximum number of stored versions
- **OO** output queue
- **PCS** personal communications services
- **PVM** parallel virtual machine
- **SPEEDES** Synchronous Parallel Environment for Emulation and Discrete-Event Simulation
- **SPN** stochastic Petri nets
- **SSR** state save rate
- **SV** state versions
- **TB** time basic (Petri nets)
- **TPN** timed Petri nets
- **TWCE** Time-Warp control engine

---

**Figure 1.** Organization of a sequential-simulation control machine.
A key factor of the DART approach is a smooth transition from the event-
driven to the time-driven paradigm, with application actors that remain unaware
of the specific control discipline enforced by the control machine. This also favors
modularity, because it lets us reuse actors to fit different application requirements.

With an object-oriented language such as Oberon-2, C++, or Java, we can
effectively program the control machine and scheduler actor using the basic
mechanisms of dynamic binding, polymorphism, and runtime type identification.
The resulting actor programming style is safe, clean, and sound.

**Distributed simulation and Time Warp**

The simulation of complex, asynchronous, and discrete event systems—such as formal models of large personal communications systems—can be very expensive in terms of the required computational resources on a single-processor machine. We can potentially accelerate such simulations by using parallel architectures (tightly coupled multiprocessors) or distributed architectures (loosely coupled multicomputers). To do this, we allocate different parts of the modeled system—logical processes, or LPs—to different processors, where a sequential simulation engine simulates them locally. Concurrent execution of messages related to different points in the simulation on different LPs or processors can potentially speed up the simulation. However, this also introduces the need for synchronization protocols, which are at the heart of the parallel or distributed simulation problem, and which can
nally. Under aggressive cancellation, the LVT immediately sends all the anti-messages as a prompt undo request. An antimessage gives rise to a rollback in the receiving LP if it has already processed the corresponding positive message.

Lazy cancellation refrains from sending all the antimessages. By this technique, the original LP only sends an antimessage if, during its new forward computation, it does not regenerate the corresponding positive message. Both techniques have demonstrated relative performance virtues in different application domains.

One key characteristic of Time Warp is its necessary frequent saving of the LPs’ states—checkpointing. Global virtual time is the simulation time committed so far for the whole simulation model. No LP can roll back to a time prior to the GVT. Consequently, each LP can free its memory space of outdated states, a process we call fossil collection.

A critical issue with Time Warp is how frequently it should update the GVT. Updating with high frequency conserves memory but wastes real time; low frequency can result in an out-of-memory exception.

Now let’s turn to a specific Time-Warp mechanism that we achieved using C++ and PVM. We split a simulation model into a collection of subsystems and allocate each to a distinct physical processor. A subsystem consists of a pair <LP, TWCE>, where LP is a logical process and TWCE is a Time-Warp control engine. LP, which is actor-based, is assigned a topological part of a timed Petri nets model. (We’ll discuss this in detail later.) TWCE includes data structures necessary for the Time-Warp operation, such as those for coping with the synchronization problem caused by stragglers. Interacting subsystems are linked by logical FIFO channels that carry intersubsystem messages, such as those used for communicating the arrival of a token to a remotely located place.

A TWCE component relies on the following basic data structures:

- The input queue (IQ) holds undos for messages that were received from external sources.
- The output queue (OQ) holds undos for messages that support a smart, aggressive cancellation technique.
- State versions (SV) stores copies of an LP’s state at different virtual times. The LP’s state essentially reduces to application actor states.

Besides its sender LP, receiver LP, and data component, each normal message includes a time stamp consisting of two timing attributes: the send time (ts) and the receive time (tr). Send time coincides with the LVT of the sender at message transmission time. Receive time represents the time at which the receiver LP should dispatch the message to its target actor. An undo message carries a single timing attribute—undo time (tu). This represents the send time on the basis of which the LP that receives the undo message must cancel all future messages previously sent by the sending LP.

The control engine ranks scheduled messages into the IQ according to their receive time, and the messages remain there after being processed. The IQ’s current position advances after a message dispatch. The control engine removes messages from the IQ during a fossil collection phase or during a cancellation phase requested by an undo message.

If an LP sends at least one message to a partner LP at time LVT*, then LP’s OQ contains an entry holding an undo message for LP with an undo time of LVT*. This organization differs from classical Time-Warp implementations, in which the OQ maintains a copy of each message sent out, but with a negative sign attached—that is, an antimessage. The OQ structure we adopted conserves memory, simplifies the OQ’s management, and supports a variation of the aggressive cancellation technique, which we discuss later.

Figure 2 presents the operation of the Time-Warp control algorithm in pseudocode. The algorithm is influenced by two simulation parameters: the state save rate (SSR) and the maximum number of stored versions (MNSV). An SSR value of 55

diminish the actual achieved speedup. Such protocols must ensure the ordering of events with respect to virtual time as in the sequential simulation, thus preserving the causality of events.

Broadly speaking, synchronization mechanisms fall into two categories—conservative and optimistic. A conservative protocol prevents causality errors from occurring by blocking an LP should there be any chance of its processing an "unsafe" event—one for which causal dependencies are still pending. An optimistic protocol, using a detection-and-recovery approach, lets an LP redo the simulation of an event should it detect that premature processing of local events is inconsistent with causality conditions generated by other LPs.

Time Warp is an optimistic strategy for distributed discrete event simulation. Under this strategy, every LP has a local clock that keeps track of local virtual time, or LVT, and a separate message queue that drives a local simulation algorithm. LPs proceed asynchronously, and thus they might have different LVT’s at the same real time. The basic synchronization problem that Time Warp must address arises when an LP receives an external message whose time stamp is lower than the LVT (we call such messages stragglers). To avoid causality errors—the future cannot influence its past—Time Warp must roll back the LP’s state to a virtual time less than or equal to the straggler’s time stamp, a process we call fossil collection.

A critical issue with Time Warp is how frequently it should update the LVT. Updating with high frequency conserves memory but wastes real time; low frequency can result in an out-of-memory exception.

Now let’s turn to a specific Time-Warp mechanism that we achieved using C++ and PVM. We split a simulation model into a collection of subsystems and allocate each to a distinct physical processor. A subsystem consists of a pair <LP, TWCE>, where LP is a logical process and TWCE is a Time-Warp control engine. LP, which is actor-based, is assigned a topological part of a timed Petri nets model. (We’ll discuss this in detail later.) TWCE includes data structures necessary for the Time-Warp operation, such as those for coping with the synchronization problem caused by stragglers. Interacting subsystems are linked by logical FIFO channels that carry intersubsystem messages, such as those used for communicating the arrival of a token to a remotely located place.

A TWCE component relies on the following basic data structures:

- The input queue (IQ) holds undos for messages that were received from external sources.
- The output queue (OQ) holds undos for messages that support a smart, aggressive cancellation technique.
- State versions (SV) stores copies of an LP’s state at different virtual times. The LP’s state essentially reduces to application actor states.

Besides its sender LP, receiver LP, and data component, each normal message includes a time stamp consisting of two timing attributes: the send time (ts) and the receive time (tr). Send time coincides with the LVT of the sender at message transmission time. Receive time represents the time at which the receiver LP should dispatch the message to its target actor. An undo message carries a single timing attribute—undo time (tu). This represents the send time on the basis of which the LP that receives the undo message must cancel all future messages previously sent by the sending LP.

The control engine ranks scheduled messages into the IQ according to their receive time, and the messages remain there after being processed. The IQ’s current position advances after a message dispatch. The control engine removes messages from the IQ during a fossil collection phase or during a cancellation phase requested by an undo message.

If an LP sends at least one message to a partner LP at time LVT*, then LP’s OQ contains an entry holding an undo message for LP with an undo time of LVT*. This organization differs from classical Time-Warp implementations, in which the OQ maintains a copy of each message sent out, but with a negative sign attached—that is, an antimessage. The OQ structure we adopted conserves memory, simplifies the OQ’s management, and supports a variation of the aggressive cancellation technique, which we discuss later.

Figure 2 presents the operation of the Time-Warp control algorithm in pseudocode. The algorithm is influenced by two simulation parameters: the state save rate (SSR) and the maximum number of stored versions (MNSV). An SSR value of 55
initialization: LVT=0; GVT=0; max_simulation_time=...;
MNSV=...; SSR=...; ...
while( GVT<max_simulation_time ){
if( an external message m was received ){
    switch( m ){
    case normal_message: schedule m at time m.t, onto 
        IQ; break;
    case undo_message: Undo( m ); break;
    case GVT_update_message: GVT_Update();
    }
}
if( IQ is not empty ){
    let next be the most imminent message from IQ;
    if( next.t, ≥ LVP ){
        LVT=next.t,;
        dispatch next to its target actor;
        for( each generated message gm ){
            if( gm is a local message )
                schedule ge at time gm.t, on IQ;
            else( //gm is an external output message
                route gm to its destination LP dest_LP;
                if( no message has been sent to dest_LP at LVT )
                    store an undo message with t,=LVT on OQL[LVT];
            }
        }
    if( LVT changed SSR times from last checkpoint )
        Checkpoint();
    else( //next is a straggler
        Rollback( next.t, );
    }
}  } //while

Figure 2. Pseudocode of the Time-Warp operation.

LP status just before the ith LVT change from the last saved version. An MNSV value of indicates that the algorithm will invoke the GVT update protocol after it has stored j state versions and the need arises to checkpoint the LP again.

The algorithm launches the rollback at time t, in SV with a checkpoint time prior to the rollback time (t, ≤ t,). It first looks for a state version in SV with a checkpoint time prior to the rollback time (t, ≤ t,), and installs that state on the rollback LP. The procedure continues by coasting forward from t, to t, in SV. During this phase, LP recovers local messages, but the algorithm disables external message transmission to avoid sending duplicate messages. The algorithm uses our modified aggressive cancellation technique, MAC, to cancel erroneous computations previously triggered by messages that the rolled-back LP transmitted to partner LPs after the rollback time (t, ≥ t,). To each partner LP, MAC sends a single undo message with the undo time set at the earliest time at which the rolled-back LP transmitted a message to the LP—that is, t, ≥ t,. After the cancellation phase, the rolled-back LP resumes normal activity by processing the straggler and scheduling the in-transit messages received during rollback.

The GVT update protocol is started by an LP that sends all the other LPs a GVT update message stating that it will not send them further messages until the end of the GVT update protocol. Each LP responds by stopping the message loop and broadcasting a similar message to all the other LPs. Once each LP has received replies from all the other LPs, all in-transit messages are eventually received and scheduled. Then, each LP calculates and sends a proposed GVT value to a designated GVT manager. Each LP calculates this value as min(LVT, IQ(first_message, t,))—that is, whichever is lower the LVT or the receive time of the first message in the IQ. The manager establishes the GVT as the least of all the proposed values and broadcasts the new GVT value to all the LPs.

Finally, the LPs resume normal operation by first collecting the memory of LP status in SV associated with times prior to the GVT. (Actually, the LP maintains the state at GVT or at a time just before GVT in case it has to support a rollback at GVT.) They also collect and discard no-longer-useful undo messages in the IQ.

Compared to classical aggressive cancellation based on antime-messages, MAC minimizes the bookkeeping involved in the Time-Warp kernel and the transmission times of antime-messages. A single antime message transmitted to each partner LP replaces the transmission of multiple antime-messages that carry, along with the negative sign, the same data component of the original positive messages. By reducing the size and number of messages exchanged among the LPs, MAC makes it possible to increase the computation granularity.

As a further benefit, MAC stops erroneous computation more quickly because it must send only a single message on the network. This is particularly effective in a distributed context, where high transmission latency of messages delays the system's ability to stop the spread of erroneous computation. Consider, for example, self-driving applications such as the simulation of PCSs. When a remote processor receives an incorrect message, the error spreads very rapidly to many other objects within the same processor because no network interprocess communication is required, and the antime-messages are delayed.

Distributed simulation of TPNs

Timed Petri nets easily allow modeling and analysis of dynamic discrete event systems (see the sidebar, "Timed Petri nets" for background and details). Basically, the net structure directly expresses causality of events, whereas the association of firing delays (that is, events) with transitions models dynamic behavior.
Timed Petri nets

Petri nets are widely used as a modeling tool for studying asynchronous concurrent systems. They have various applications in science and engineering—for instance, in modeling distributed software systems or communication networks and protocols.

We can verify certain properties of Petri nets by relying on theoretical results. For instance, structural and reachability analysis lets us answer qualitative questions about the modeled system’s liveness, boundedness, invariants, and other characteristics. We can perform the analysis by applying linear algebra techniques or by investigating the set of reachable states.

On the other hand, we can evaluate quantitative characteristics by exploiting the notion of time explicitly added to the classical definition of Petri nets. Relating time to the firing delay of a transition, Petri net models extended by timing are generally called timed Petri nets or TPNs.

TPNs basically can associate time with either places or transitions. The two approaches have been proven equivalent, in the sense that we can represent a model in one approach with a model in the other approach. For this article, we chose the class of TPNs that associate timing information with transitions.

Definition of Terms

A TPN is a tuple \((P, T, F, W, \Pi, \tau, M)\),

- \(P \subseteq \{p_1, p_2, ..., p_n\}\) is a finite set of \(P\) elements, or places,
- \(T \subseteq \{t_1, t_2, ..., t_m\}\) is a finite set of \(T\) elements, or transitions, with \(P \cap T = \emptyset\) and a nonempty set of nodes, \(P \cup T \neq \emptyset\).
- \(F \subseteq (P \times T) \cup (T \times P)\) is a finite set of arcs between \(P\) elements and \(T\) elements.
- \(W : F \rightarrow N\) assigns weights \(w(f)\) to elements of \(f \in F\) denoting the multiplicity of unary arcs between the connected nodes.
- \(\Pi : T \rightarrow N\) assigns priorities \(\pi\) to \(T\) elements \(t \in T\).
- \(\tau : T \rightarrow \mathbb{R}\) assigns firing delays \(\tau\) to \(T\) elements \(t \in T\).
- \(M: P \rightarrow N^0\) is the marking \(M(t) = \sum p_i \in P\) of \(P\) elements \(p_i \in P\).

There are three important cases of firing delays:

- immediate transitions, when \(\tau = 0\),
- deterministic timed transitions, when \(\tau \in \mathbb{R}\) is a deterministic time value, and
- stochastic timed transitions, when \(\tau\) is an instance of a random variable.

If \(T\) contains only stochastic timed transitions where the firing delay random variable is exponentially distributed, \(T\) belongs to the class of stochastic Petri nets. Generalized SPNs (GSPNs) allow a combination of nonstaged (immediate) and stochastically timed transitions.

Timed enabling and firing semantics

- Let \(I(t)\) and \(O(t)\) denote the set of input and output places of \(t \in T\). A transition \(t \in T\) is enabled in some marking \(m\) at time \(t\) if and only if \(\forall p \in I(t), (\mu(p) \geq w(p) \sigma(0)) m\).
- If \(E(\mu)\) contains only timed transitions, and \(R(T)\), \(t \in E(\mu)\) is the remaining enabling time of \(t\) during which the enabling tokens reside in \(T\) and \(t\) is the transition that must fire next is the one with \(t \in T\) enabled.
- If \(\forall p \in I(t), (\mu(p) \geq w(p) \sigma(t)) \land c(t) = 1\), then \(t\) is said to be enabled at degree \(c\). It can only fire one enabling at a time, it adopts the single-server semantics. Infinite server semantics occurs when any amount of enablings can be fired at one time, expressing a notion of parallelism among them.

There are a variety of different firing rules defined upon TPNs. In the SPN class, for example, transition firing is atomic. A random time elapses between the enabling and the firing of a transition \(t\), during which the enabling tokens reside in the input places. Transition \(t\) must be continuously enabled during the time \(t\) and it fires at that time: this is the race or preemptive policy. Another execution rule for TPNs occurs when an enabled transition fires in three phases: in a “start firing” phase, tokens are removed from the input places, remaining invisible during the “firing in progress” phase until they are released into output places in the “end firing” phase. This is the preselection or nonpreemptive policy.

References


It is possible to instrument the execution of a TPN model on a single machine—this is called sequential simulation. However, to animate complex systems, parallel or distributed simulation are preferable. Next, we present a DART-centered, modular approach for the distributed execution and simulation of TPNs.

To improve performance, we can partition a TPN model into several regions so that conflicting transitions together with all their input places reside in the same LP. An obvious mapping of TPNs on our Time-Warp architecture would be to associate distinct actors to transitions and places of a TPN, with an initial
configuration that creates all the actors of a region and initializes them according to the region topology using acquaintance relations. Transition and place actors would interact by suitable messages during conflict resolution of enabled transitions and firings. However, we chose a different solution. Our goals were to

- allow for the configuration of coarse-grained regions adequate for a distributed system of standard workstations;
- minimize the number of exchanged messages, a critical factor for the simulation of complex systems; and
- facilitate the region's (LP) status-saving and -restoring operations necessary to the Time-Warp mechanism.

For each LP, we introduce a region actor initialized with a passive data structure describing the region topology (see Figure 3). The region implements the functional TPN execution model. The internal status of a region actor includes a marking vector that has an entry for each place in the region, and a firing vector that registers the history of transition firings for statistical data collection. A marking entry simply stores the number of tokens present in the associated place. An enabling entry stores the number of available enablings for the associated transition. This can be zero if the transition is disabled or a value $c$ if the transition is multiply enabled with degree $c$.

A metaregion actor is fed transition attributes—that is, the mean of the exponential distribution function for timed transitions and a priority value for untimed transitions. An unspecified priority defaults to a standard priority value. The metaregion transparently filters region messages and applies to them a suitable policy that ultimately affects scheduling.

Region and metaregion constitute a synergic tandem. They must agree on some fundamental points, such as whether they will use preemption or preselection as the strategy regulating transition enabling and firing. For generality, we adopt by default the infinite-server semantics for transition firing. If we needed single-server semantics, we could add to each transition a loop-back place with a single assigned token.

Following the firing of a transition $t$, the region actor, under race conditions, must identify the set of transitions that have increased their enablings and the set of transitions that have lost some enablings. For each new enabling or disabling, the region generates and sends to itself a distinct $\text{transition}_\text{fire/unfire}$ message with the associated transition as a parameter. Under preemption, conflicts are automatically solved at runtime—that is, at transition firings. A $\text{transition}_\text{unfire}$ message requires the metaregion to unschedule a previously scheduled $\text{transition}_\text{fire}$ message. Processing a $\text{transition}_\text{fire}(t)$ consists of the following steps:

1. The region modifies the firings vector by annotating the firing of transition $t$ and, possibly, its occurrence time.
2. The region modifies the marking vector by removing an enabling tuple from the input places of transition $t$ and adding the required tokens into output places of $t$. If an output place belongs to a remote LP or region, the region generates a corresponding $\text{token}_\text{arrival}$ message.
3. The region modifies the enabling vector by examining the enablings of the transitions whose input places were influenced by the firing of $t$. It generates a corresponding group of $\text{transition}_\text{fire/unfire}$ messages on the basis of variations between old and new values of enablings.

A region processes a $\text{token}_\text{arrival}$ message using steps similar to the marking update (step 2) and the enabling update (step 3).

The metaregion schedules untimed $\text{transition}_\text{fire}$ messages according to transition priority, or nondeterministically if the priority is the same. It normally schedules timed $\text{transition}_\text{fire}$ messages according to the transition's next-sample fire time, determined through an exponential variate generator.

Under preselection, the metaregion performs a preliminary ranking of the just-sent $\text{transition}_\text{fire}$ messages on the basis of transition attributes; it tentatively schedules them according to this order. However, it actually schedules a $\text{transition}_\text{fire}$ message provided it has not been disabled by a previously scheduled $\text{transition}_\text{fire}$. If the transition is still enabled, the metaregion first removes its enabling tuple from its preset and considers its firing committed. If the transition has been disabled, the metaregion simply abandons the $\text{transition}_\text{fire}$ message.

To support metaregion operation, the region actor exports two synchronous accessor methods: $\text{enabled}(t)$ and $\text{preselect}(t)$. The former determines whether transition $t$ is enabled. The latter preselects $t$'s enabling tokens. Processing a $\text{transition}_\text{fire}$ message follows the same steps as in the preemption case, but step 2 reduces to the generation of tokens in the output places. Obviously, this case does not require $\text{transition}_\text{unfire}$ messages.

Further specializations of the metaregion are possible. For example, we could customize the preemption policy with an aging policy. The metaregion could save the remaining time of a transition whose enabling was interrupted and use it as the fire time of the next enabling instead of evaluating a new random sample.

The separation of concerns between region and metaregion permits the execution model of TPNs to be sensitive to
different timing models—deterministic models, for example. We can also exploit this separation in the important case of supporting the execution and timing analysis of TPN formalisms for hard real-time systems, such as TB nets.\textsuperscript{12} Finally, because the region and metaregion paradigm cannot possibly annotate transitions, the resulting behavior under preemption reduces to that of ordinary Petri nets. We can use this characteristic to prove qualitative properties of a modeled system.

**Case study using large PCS networks**

Now we'll turn to a TPN model for a personal communication services network—a cellular communication system providing voice services to mobile subscribers.\textsuperscript{13}\textsuperscript{14} The service area has a wrap-around, Manhattan-like topology partitioned into regular subareas called cells. Service within a cell comes from a base station identified by its \(xy\) coordinates. Figure 4 shows the behavior of a mobile user, modeled as a stateless token. The figure portrays a TPN model of a generic cell according to a preemption strategy. We achieve an actual system topology by spatial replication of the basic cell model, a topic we'll cover later in this section.

Each cell is assigned a fixed number of channels. Movement and call issues are orthogonal to one another. A user can enter a cell with or without a call in progress. If a call is on, the user token releases the old channel to the exiting cell and requests a new channel from the entering cell; this is called the hand-over procedure. If no channel is available, the call enters a waiting queue—a buffer—where it remains pending for a fixed amount of time (this is just an example of a possible policy). If a channel does not become available by the end of the waiting period, the network considers the call blocked. Otherwise the call continues with a newly assigned channel. Of course, a user can initiate a new call from within a cell; the network handles this exactly as it does a hand over. In the following discussion, we use the term *channel request* for both a hand over and a new call. Quality of service can require designing the system (the number of channels per cell, for example) to keep the blocking probability below a given value, such as \(10^{-3}\).

Figure 4 portrays timed transitions as rectangles. A user with a call in progress is received into the call-request place, where the hand-over procedure begins. A user without a call in progress is received into the call-off place. From call-off, the user token can initiate a new call (transition new-call) or abandon the cell (transition exit-as-off). If the user token tries a new call, it moves into the call-request place. A satisfied channel request (transition start-call1) moves the user token into the call-on place. If a channel is not available (transition call-pending), a user token passes from call-request into the wait-queue place, where the channel request remains pending. A waiting request can be preempted by the arrival of a new channel (transition start-call2). Otherwise (transition blocked), the network considers the call blocked, and the token moves from the wait-queue place to the call-off place; this happens when a call is rejected or a hand over fails. In the call-on state, the user can abandon the cell (transition exit-as-on) or terminate the call (transition end-call), moving to the call-off place. In both cases, the token first releases the channel it used to the channel pool. Out-on and out-off are exiting places for users with or without a call in progress. From an output place, a user can nondeterministically move to one of the four neighboring cells to the north, east, south, and west. The exit transitions from out-on to incoming arcs to the call-request places in the neighboring cells; the exit transitions from out-off link to incoming arcs to their call-off places.

The system achieves statistical timed behavior by timed transitions and related exponential means. Exit-as-off transitions have a mean dwell time of \(1/\delta\); new-call transitions have a mean call interarrival time of \(1/\lambda\); and end-call transitions have a mean call duration time of \(1/\mu\). Blocked transition is instead bound to a fixed delay time \(\Delta\). The system achieves nondeterminism among input/output \(W, N, E, \text{ and } S\) immediate transitions by attaching the default priority to them.

We can easily use the cell TPN model to collect statistical data of the physical system. For example, we can estimate the blocking probability by counting the total number of firings of the blocked transition and dividing it by the total number of firings of the blocked, start-call1 and start-call2 transitions.

For space considerations, we do not depict a TPN model equivalent to that in Figure 4 but using the preselection firing strategy. However, we will make a few remarks about such a model. A request that does not find an available channel (transition call-pending) is duplicated in the timed-queue and queue places. From timed-queue, a channel

![Figure 4](image-url)
request is extracted after Δ time units and deposited into the decision place. A buffered request that receives a channel before its waiting time has expired (transition start-call2) becomes a call on (at the call-on place) and is also duplicated into the served place. An expired channel request in the decision place counts as a blocked call (transition blocked) when it is not served by a channel during the waiting time. Otherwise, it is simply canceled (the drop transition).

In the next section, we report some experimental results that we collected using the preselection model.

**Simulation Results**

Christopher Carothers and colleagues show that it takes at least 256 cells to avoid the wrapping effect derived by the chosen topology. (The wrapping effect refers to the simulation error due to boundary cells that "wrap" back on themselves. A mobile unit at an edge suddenly moves and reappears at the other side of the grid of cells, thus causing errors in the call statistics. This error can be reduced by using a larger network of cells.) Accordingly, we set the actual simulated PCS system with the following parameters:

- Number of base stations: 400 (a square of 20 cells x 20 cells);
- Dwell time 1/6: 10 seconds;
- Call duration 1/670 seconds;
- Call interarrival time 1/6: 140 seconds;
- Number of users per cell: 7 and 25 (2,800 and 10,000 users in the whole system); and
- Number of users per channel: 3.5.

We performed the simulation experiment using a variable number \( P \) of Rice6000 workstations dedicated to the simulation and connected by a Fast Ethernet LAN.

First, we tuned the Time-Warp parameters MNSV and SSR empirically with the aim of minimizing completion time. Figure 5a shows the completion time as a function of MNSV for SSR set at 0 and 7 users per cell. A low value for MNSV requires each LP to stop frequently during its normal computation to synchronize constraints with the other LPs. All of this increases the system overhead, but reduces the bookkeeping activities due to the lower number of system states to manage.

The SSR parameter lets us tune the checkpoint interval. Frequent state saving decreases rollback costs by reducing the rollback length, but augments the time required to perform state-saving operations during normal activity. Figure 5b illustrates the influence of SSR on completion time.

We conducted both the preparatory tests illustrated in Figure 5 by running the PCS model on four processors for a simulation period of 1,000 time units (seconds). We obtained the completion time by using the Unix `times` function. We carried out the remaining simulation...
work, documented in Figures 6 to 11, with SSR set at 0, MNSV set at 15, and 1,000 simulated seconds.

Because of the homogeneous character of our test network, we managed load balancing simply by defining regions as a split of the simulated square area into vertical slabs so that each region contained the same number of cells. To obtain a straightforward partitioning, we set the number of processors for the experiments to two, four, five, and 10.

In studying the effect of the number of processors $P$ on performance, we observed that each LP becomes less loaded as we increase $P$—that is, the number of events required to increase the LVT is smaller. This allows each LP to advance the LVT autonomously at a higher rate (with respect to real time), and thus the overhead of communication latency increases. In other words, the probability increases that a target LP has significantly augmented its LVT during the time interval between another LP's sending a message and the target LP's actually receiving it, so more messages are stragglers. Figure 6 confirms this, showing the percentage of external messages an LP receives that are stragglers as a function of $P$. As expected, for a fixed number of users in the system, augmenting $P$ increases the straggler percentage, too. We obtained the reported values as the mean of the values observed for a single LP.

Figure 7 depicts the net effect of the number of processors on the simulation performance, portraying speedup as a function of $P$ for the two values of the number of users.

To help clarify the speedup curves, Figure 8 portrays the measured event rate—that is, the number of committed events processed per second of real time by both the sequential and the parallel simulators. The sequential simulator uses a basic simulation engine, not Time Warp running on one processor. Under 25 users per cell, the sequential simulator processes about $14 \times 10^6$ committed events with a completion time of $10.47 \times 10^6$ seconds and an event rate of $1,350$ events per second.

Figure 9 shows the computational granularity—that is, the ratio between the total number of remote events transmitted by an LP (including retransmissions due to rollbacks) and the total number of processed events. The reported values are the mean of the values observed for a single LP. As $P$ increases, the ratio augments because of a reduction in the number of events processed by an LP, although the number of events received by an LP, although the number of events transmitted on the network to the total number of sequential committed events is almost constant. The sequential simulator processes about $14 \times 10^6$ committed events with a completion time of $10.47 \times 10^6$ seconds and an event rate of $1,350$ events per second.

Communication overhead for two, four, five, and 10 processors: the ratio of the total number of remote events transmitted on the network to the total number of sequential committed events. As $P$ increases, the ratio augments because of a reduction in the number of events processed by an LP, although the number of events received by an LP, although the number of events transmitted on the network to the total number of sequential committed events is almost constant. The sequential simulator processes about $14 \times 10^6$ committed events with a completion time of $10.47 \times 10^6$ seconds and an event rate of $1,350$ events per second.

We measured the communication overhead as the ratio of the total number of remote events transmitted on the network to the total number of sequential committed events. Figure 10 summarizes the results.

As an example of quality-of-service estimation, Figure 11 reports the blocking probability as a function of the maximum waiting time $\Delta$ for the case of seven users per cell and 3.5 users per channel. These results confirm that the higher the time a call can remain in the buffer waiting for a channel, the lower is the value of the blocking probability. Obviously, the blocking probability depends also on the number of channels assigned to each cell. Therefore, blocking performance should also be studied as a function of other system parameters—the number of channels, the number of users, and so forth. Such studies would provide support for systems dimensioning—for example, trading off the number of channels per cell and the value of $\Delta$—which can assure the requested value for blocking probability.

THE ACTOR MODEL we adopted for structuring LPS can be related to the computational model of the SPEEDES Time-Warp system. In SPEEDES, an LP is separated into a data object and one or more event objects. A data object is not aware of simulation time and hosts a local state manipulated by methods and affected by Time Warp's saving-and-restoring operations. An event object has the logic...
for accessing a data object and can schedule new event objects. This way, a data object can function in different applications with different requirements not necessarily tied to simulation. These are the same goals that actors address in our work: We split responsibilities among application (or functional) actors that encapsulate a local state manipulated by answering messages, and reflective actors that are aware of timing and perform message scheduling. The local state of application actors is sensitive to Time-Warp behavior. The two kinds of actors are glued together by the event-driven paradigm of asynchronous messages generated by application actors and ultimately handled by reflective actors.

The experimental data we collected using our system for the simulation of a TPN model for large PCS networks showed good performance. For a PCS system with 400 cells and 10,000 users, our Time-Warp system demonstrated a speedup of eight with 10 processors.

Currently, we are continuing our experiments with the approach in the simulation of complex systems. We are also specializing the region and metaregion actors to support the execution of TPN formalisms adequate for time-critical systems, thus allowing temporal validation activities.

Further directions for our work concern

- extending our Time-Warp mechanism with optimistic optimizations, notably for memory control;
- improving our PCS TPN model along different design dimensions (such as call-buffering policies during hand overs, multimedia traffic, and so forth);
- comparing our PCS simulation work with the results reported by Christopher Carolathers, Richard Fujimoto and Yi-Bing Lin—first by reproducing a plain aggressive cancellation technique and then quantifying the effects due to the variation of the aggressive cancellation we described here;
- implementing our mechanisms in Java to improve portability and move toward distributed simulation at the Web level; and
- developing a graphical tool for designing a TPN model and guiding the region partitioning and allocation issues.

The modular approach we described in this article is also currently in use in the development of distributed real-time systems and multimedia applications with a special focus on the synchronization of multistream sessions.

ACKNOWLEDGMENTS

The authors are grateful to Francesca Pupo and Giancarlo Forzano for their contribution to the work described here, and to the anonymous referees for their constructive suggestions. Work on the Time-Warp project was carried out under the financial support of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) within the framework of the project, "Methodologies and Tools for High-Performance Systems for Distributed Applications."

REFERENCES


Roberto Beraldi’s research interests include distributed systems, parallel simulation, and performance evaluation of wireless systems. In 1996, he received his PhD in computer science from the University of Calabria, Italy (Unical), graduating in distributed simulation of telecommunication systems.

Libero Nigro is an associate professor of computer science in the Department of Electronics, Informatics and Systems Science of Unical. His research interests include object-oriented software engineering and time-dependent distributed systems—real-time, simulation, and multimedia. He received the laurea degree in electrical engineering from the Engineering Faculty of Unical. He is a member of the IEEE Computer Society and the ACM. Contact him at Università della Calabria, Dipartimento di Elettronica, Informatica e Sistemistica, I-87036 Rende(CS) - Italy, Lnigro@unical.it.