# Modelling, Simulation and Implementation of Distributed DEVS Simulators

Eugene Syriani and Amr Al-Mallah

COMP 512 - Project Report

**Abstract.** Distributing the Discrete Event system Specification (DEVS) allows for increase in performance, scalability in terms of the size of the models, interoperability and resource sharing. In this report, we first model a distributed DEVS simulator as a DEVS model, to then implement it in a distributed environment. This allows for optimal estimations of parameters in a distributed DEVS implementation.

## 1 Introduction

Distributed environments overcome many of the limits imposed by large-scale tasks performed on a single processor. Modelling and simulation frameworks consume a lot of resources. In this report we consider the Discrete Event system Specification (DEVS) as a modelling and simulation framework. Our experiences show that using this formalism for large-scale tasks (such as model transformation) is very convenient.
A distributed environment for DEVS models simulation is attractive for several reasons. Although not the primary goal of distributed simulation, model execution time can be reduced. The limited memory issue for a single machine can be overcome to handle large models and thus gain a scale in performance. But the main goal of distributing DEVS is for interoperability. Handling geographically distributed users and/or resources (e.g., databases, specialized equipment) and exploiting greater data handling capability through specialized nodes is our main interest. As a side effect, this allows integrating simulations running on different platforms. Furthermore, properties of distributed systems such as fault-tolerance capabilities become accessible.
After introducing the DEVS language and its simulator in section 2, we describe the model example used for both the modelling and the implementation in section 3. the remainder of this report focuses on distributed DEVS techniques. A distributed DEVS simulator modelled as a DEVS model is described in section 4. Section 5 is devoted to the implementation of such a simulator in a distributed environment. Section 6 presents and discusses advantages and disadvantages of three current implementations of distributed DEVS architectures: `DEVS/Grid`, `DEVS/RMI`, and `DEVS P2P`. Finally we conclude and plan future work in section 7.

## 2 The Discrete Event System Specification

### 2.1 The Classical DEVS formalism

The DEVS formalism was introduced in the late seventies by Bernard Zeigler to develop a rigorous basis for the compositional modelling and simulation of discrete event systems [1]. It has been successfully applied to the design, performance analysis and implementation of a plethora of complex systems.
A DEVS block model is either an *AtomicBlock* or a *CoupledBlock*. An atomic model describes the behaviour of a reactive system. A coupled model is the composition of several DEVS sub-models which can be either atomic or coupled. Submodels have *ports*, which are connected by channels (represented by the associations between the different ports). Ports are either *Inport* or *Outport*. The abstract classes *(In/Out)port* can be instantiated in either an *Atomic(In/Out)port* or a *Coupled(In/Out)port* respectively. Ports and channels allow a model to receive and send signal events (any subclass of *Event*) from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an output port of its parent model.

An **atomic DEVS** model is a structure $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$ where $S$ is a set of sequential **states**, one of which is the *initial* state. $X$ is a set of allowed **input events**. $Y$ is a set of allowed **output events**. There are two types of transitions between states: $\delta^{int} : S \to S$ is the **internal transition function**, $\delta^{ext} : Q \times X \to S$ is the **external transition function**, Associated with each state are $\tau : S \to \mathbb{R}_0^+$, the **time-advance** function and $\lambda : S \to Y$, the **output function**. In this definition, $Q = \{(s, e) \,|s \in S, 0 \leq e \leq \tau\,(s)\}$ is called the **total state space**. For each $(s, e) \in Q$, $e$ is called the **elapsed time**. $\mathbb{R}_0^+$ denotes the positive reals with zero included. Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It will remain in any given state for as long as the time-advance of that state specifies or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) sends output as specified by *outputFunction*, and then instantaneously jumps to a new state specified by *internalTransition*. If input is received however before the time for the next internal transition, then it is *externalTransition* which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

A **coupled DEVS**[1] model named $D$ is a structure $\langle X, Y, N, M, I, Z, select \rangle$ where $X$ is a set of allowed **input events** and $Y$ is a set of allowed **output events**. $N$ is a set of **component names** (or labels) such that $D \notin N$. $M = \{M_n \mid n \in N, M_n$ is a DEVS model (atomic or coupled) with input set $X_n$ and output set $Y_n\}$ is a set of DEVS **sub-models**. $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ is a set of **influencer** sets for each component named $n$. $I$ encodes the connection topology of sub-models. $Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \to X_n \; or \; Z_{D,n} : X \to X_n \; or \; Z_{i,D} : Y_i \to Y\}$ is a set of **transfer functions** from each component $i$ to some component $n$. $select : 2^N \to N$ is the **select** or tie-breaking function. $2^N$ denotes the powerset of $N$ (the set of all sub-sets of $N$).

The connection topology of sub-models is expressed by the influencer set of each component. Note that for a given model $n$, this set includes not only the external models that provide inputs to $n$, but also its own internal sub-models that produce its output (if $n$ is a coupled model.) Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. For example, a "departure" event output of one sub-model is translated to an"arrival" event on a connected sub-model's input. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled to be performed at the same time. The modeller controls which of the conflicting sub-models undergo its transition first by means of the *select* function.

We have developed our own DEVS simulator called `pythonDEVS` [2], grafted onto the object-oriented scripting language Python.

### 2.2 The DEVS simulation protocol

This section is based on Prof. Vangheluwe's lecture notes for a modelling and simulation course at McGill University.

The algorithm in Fig. 1 is based on the closure under coupling construction and can be used as a specification of a -possibly parallel- implementation of a DEVS solver or "abstract simulator" [1]. In an atomic DEVS solver (a.k.a., simulator), the last event time $t_L$ as well as the local state $s$ are kept. In a coordinator (the coupled DEVS solver), only the last event time $t_L$ is kept. The next-event-time $t_N$ is sent as output of either solver. It is possible to also keep $t_N$ in the solvers. This requires consistent (recursive) initialization of the $t_L$s. If kept, the $t_N$ allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator involves handling four types of messages. The $(x, from, t)$ message carries external input information. The $(y, from, t)$ messages are used for scheduling (synchronizing) the abstract simulators. The $(*, from, t)$ and

| message $m$ | simulator | coordinator |
|---|---|---|
| $(*, from, t)$ | simulator correct only if $t = t_N$ | |
| | $y \leftarrow \lambda(s)$<br>**if** $y \neq \phi$ :<br>  **send** $(\lambda(s), self, t)$ to *parent*<br>$s \leftarrow \delta_{int}(s)$<br>$t_L \leftarrow t$<br>$t_N \leftarrow t_L + ta(s)$<br>**send** $(done, self, t_N)$ to *parent* | **send** $(*, self, t)$ to $i^*$, where<br>  $i^* = select(imm\_children)$<br>  $imm\_children = \{i \in D \mid M_i.t_N = t\}$<br>$active\_children \leftarrow active\_children \cup \{i^*\}$ |
| $(x, from, t)$ | simulator correct only if $t_L \leq t \leq t_N$ (ignore $\delta_{int}$ to resolve a $t = t_N$ *conflict*) | |
| | $e \leftarrow t - t_L$<br>$s \leftarrow \delta_{ext}(s, e, x)$<br>$t_L \leftarrow t$<br>$t_N \leftarrow t_L + ta(s)$<br>**send** $(done, self, t_N)$ to *parent* | $\forall i \in I_{self}$ :<br>  **send** $(Z_{self,i}(x), self, t)$ to $i$<br>  $active\_children \leftarrow active\_children \cup \{i\}$ |
| $(y, from, t)$ | | $\forall i \in I_{from} \setminus \{self\}$ :<br>  **send** $(Z_{from,i}(y), from, t)$ to $i$<br>  $active\_children \leftarrow active\_children \cup \{i\}$<br>**if** $self \in I_{from}$ :<br>  **send** $(Z_{from,self}(y), self, t)$ to *parent* |
| $(done, from, t)$ | | $active\_children \leftarrow active\_children \setminus \{from\}$<br>**if** $active\_children = \emptyset$:<br>  $t_L \leftarrow t$<br>  $t_N \leftarrow min\{M_i.t_N \mid i \in D\}$<br>  **send** $(done, self, t_N)$ to *parent* |

**Fig. 1.** DEVS simulation procedure

$t \leftarrow t_N$ of topmost coordinator
**repeat until** $t \geq t_{end}$ (or some other termination condition)
  **send** $(*, main, t)$ to topmost coupled model $top$
  **wait** for $(done, top, t_N)$
  $t \leftarrow t_N$

**Fig. 2.** DEVS simulation mainloop

$(done, from, t)$ messages are used for scheduling (synchronizing) the abstract simulators. In these messages, $t$ is the simulation time and $t_N$ is the next-event-time. The $(*, from, t)$ message indicates an internal event $*$ is due. When a coordinator receives a $(*, from, t)$ message, it selects an imminent component $i^*$ by means of the tie-breaking function *select* specified for the coupled model and routes the message to $i^*$. Selection is necessary as there may be more than one imminent component (with minimum next remaining time).
When an atomic simulator receives a $(*, from, t)$ message, it generates an output message $(y, from, t)$ based on the old state $s$. It then computes the new state by means of the internal transition function. Note how in DEVS output messages are only produced while executing internal events. When a simulator outputs a $(y, from, t)$ message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation, to each of the influencees of $i^*$ (if any). If the coupled model itself is an influencee of $i^*$, the output, after appropriate output-to-output translation, is sent to the coupled model's parent coordinator.
When a coordinator receives an $(x, from, t)$ message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components.
When an atomic simulator receives an $(x, from, t)$ message, it executes the external transition function of its associated atomic model.
After processing an $(x, from, t)$ or $(y, from, t)$ message, a simulator sends a message to its parent coordinator to prepare a new schedule. When a coordinator has received $(done, from, t_N)$ messages from all its components, it sets its next-event-time $t_N$ to the minimum $t_N$ of all its components and sends a $(done, from, t_N)$ message to its parent coordinator. This process is recursively applied until the top-level coordinator or `root coordinator` receives a $(done, from, t_N)$ message.
To run a simulation experiment, the `initial conditions` $t_L$ and $s$ must first be set in all simulators of the hierarchy. If $t_N$ is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop described in Fig. 2 is executed.

### 2.3 Parallel DEVS

The Parallel DEVS formalism [3] has been introduced in order to allow collision handling and parallelism of DEVS models. Classical DEVS and Parallel DEVS formalisms only have slight modifications.
A confluent transition function $\delta^{con} : Q \times X \rightarrow S$ is added to the classical atomic model. It is triggered when the collision behaviour of an atomic block receiving an external event at the time of its internal transition. In pyDEVS it is always the external transition that happens. The event set $X$ is now a set of bags of events since atomic simulators may output events concurrently. As for the coupled model, the *select* function is removed since all delta functions run in parallel. The simulator-coordinator protocol is thus adapted to Parallel DEVS as described in section 3 of [4].

## 3 A Client-Server Example

The running example of this report is a simple client-server model. The server consists of three processors in parallel with the following description:

- Each processor has a finite FIFO queue of length n, where unprocessed jobs can be stored.
- As soon as a processor becomes idle (the current job is finished), it starts processing the first message in its queue.
- Jobs are sent by a client with a generation time-interval randomly chosen from a uniform distribution.
- Jobs are characterized by their size which corresponds to the amount of time the jobs require to be processed. Sizes are randomly chosen randomly.
- The client sends its job to the first available processor (a processor is available if it is idle or if its queue is not full), otherwise the job is discarded.
- The server notifies the client when a job is completed.

As illustrated in Fig. 3, the *System* is modelled as a coupled DEVS model. The *Client* atomic model sends job requests through its *request* outport and receives notifications via its *reply* inport. When the *Server* coupled model receives a job through its *in* inport, the job is instantaneously sent to *Processor 1*. If its state is *IDLE* and has an empty queue, it processes the job and outputs an acknowledgement via its *out* outport. The channel from the *out* ports to the *client*'s inport ensure reception of the notification. In the case where *Process 1* is not idle, it queues the job. If there is no space left for the size of that job, *Processor 1* transfers the job to *Processor 2* by the *discard* outport. The behaviour of the other two processors is identical. However, if *Processor 3* discards a job, this job is lost (hence the client is not acknowledged).

As a case-study, we would like to execute this DEVS model in a distributed environment. For this example, we propose to distribute the simulators (not the model). An *atomic solver* is associated with each atomic model and a *coordinator* with each coupled model. The distributed environment is composed of a cluster consisting of two machines *Machine 1* and *Machine 2*. Initially, the simulators are partitioned as follows: the system coordinator and the client solver are launched on *Machine 1* and the server and its processors on *Machine 2*. The *root coordinator*, which ensures the execution of the simulators, runs on *Machine 1*.

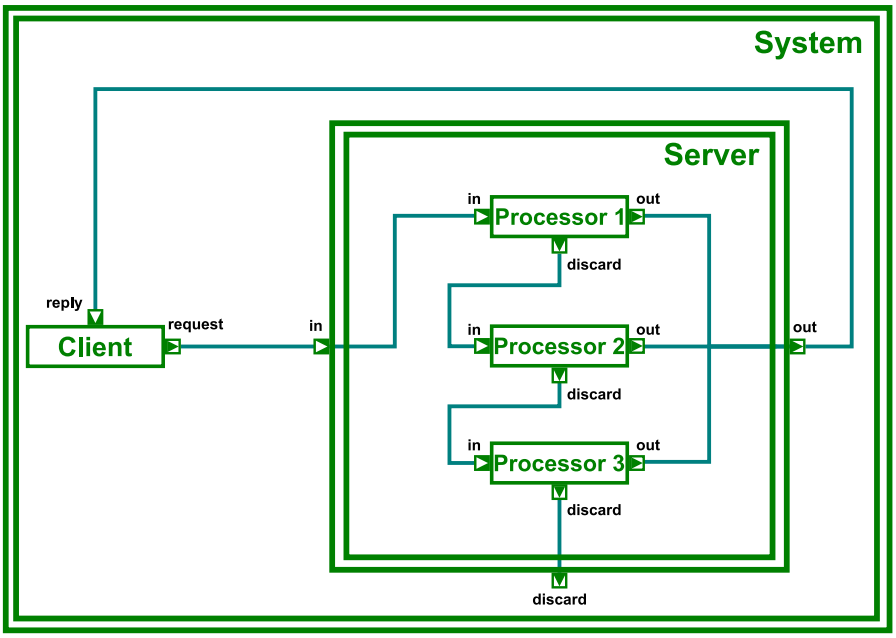


**Fig. 3.** The client-server model

# 4 Modelling a distributed DEVS Simulator

In this section, we build a model representing a distributed simulation engine for the DEVS model described in section 3. The model takes into consideration the simulation entities, the different machines as well as the

network commutation of the nodes. Fig. 4 illustrates the simulation entities modelled in DEVS representing the distributed simulation of the client-server model.

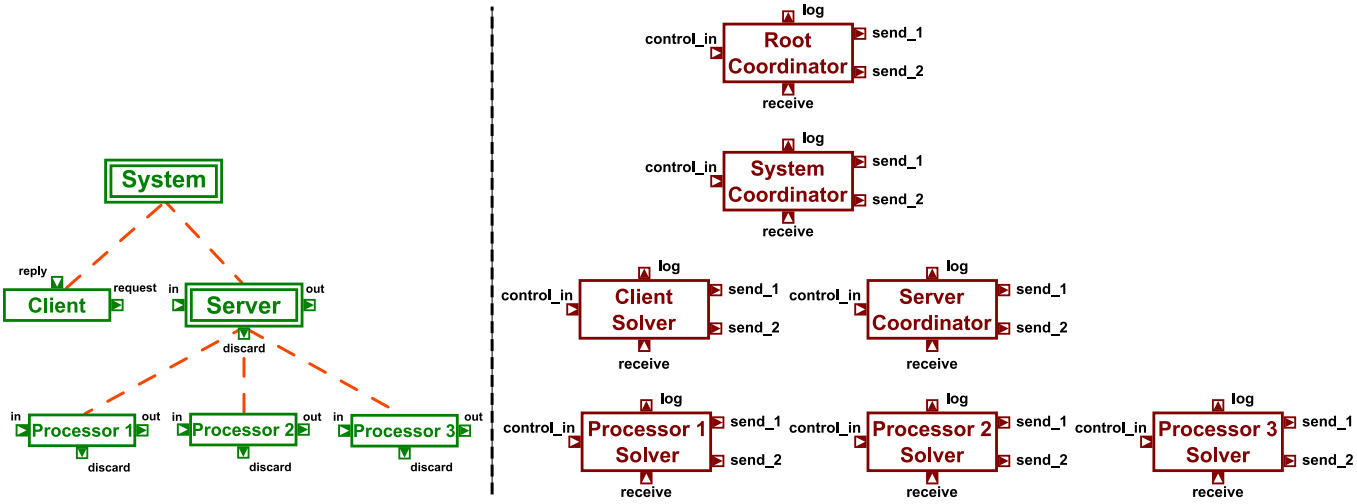


**Fig. 4.** The hierarchical model of client-server example (on the left) and and its corresponding simulation entities as DEVS models for the distributed simulation (on the right)

The notation used in this section is the same as in section 2.

### 4.1 The simulation entities

This model consists of seven atomic models ensuring the simulation of each of the six components of the client-server model. Solvers and coordinators hold their corresponding model in the state. E.g., *Client Solver* holds the *Client* atomic model and *Processor Solver 1,2, and 3* hold their respective *Processor 1,2, and 3* atomic model. *System Coordinator* holds the *System* coupled block and *Server Coordinator* holds the *Server* coupled block. The *Root Coordinator* is modelled as a separate atomic model.
There are four types of events that these atomic models exchange:

- `*`: $(source, target, t)$ message to indicate that an internal event $*$ is due,
- `X`: $(x, source, target, t)$ message to carry external input information $x$,
- `Y`: $(y, source, target, t)$ message to carry output information $y$, and
- `DONE`: $(source, target, t)$ message to acknowledge termination of event handling.

**The Atomic Solver model** (AS) has one inport RECEIVE and as many SEND outports as there are machines. In our case the outports are SEND1 and SEND2. Its state is composed of:

- the atomic DEVS model $M$ it simulates,
- a unique identifier $id$ in a one-to-one correspondence with $M$ given by the bijection `map`,
- its parent identifier $parentId$,
- the last event time $t_L$,

– the next event time $t_N$,
– the output set $\Lambda$, and
– a single-rowed table `activeOutport` identifying which outport is currently active, and
– the mode $\mu$ the AS is in: *PAUSED* or *RUNNING*.

AS is reactive, waiting for a `*` or `X` external event as described in Algorithm 1. It also receives an initialisation message `INIT` to which its sets its state. AS can also receive simulation control event $\chi$ from the CONTROL_IN port to set its mode.

**Algorithm 1** The external transition $\delta_{ext}\left((s,e),x\right)$ of an atomic solver

**UPON RECEIVE** `INIT`:$(parent, t'_L, M', s'_M, \Lambda', \texttt{activeOutport'})$ **then**
$M \leftarrow M'$
$s_M \leftarrow s'_M$
$parentId \leftarrow parent$
$t_L \leftarrow t'_L$
$t_N \leftarrow t_L + \tau^M(s_M)$
$\Lambda \leftarrow \Lambda'$
`activeOutport` $\leftarrow$ `activeOutport''`
$\mu \leftarrow PAUSED$

**UPON RECEIVE** `X`:$(x, source, target, t)$ **then**
$s_M \leftarrow \delta^M_{ext}\left((s_M, t - t_L), x\right)$
$t_L \leftarrow t$
$t_N \leftarrow t_L + \tau^M(s_M)$
$M.elapsed \leftarrow 0$

**UPON RECEIVE** `*`:$(source, target, t)$ **then**
$\Lambda \leftarrow \Lambda \cup \{\lambda^M(s_M)\}$
$s_M \leftarrow \delta^M_{int}(s_M)$
$t_L \leftarrow t$
$t_N \leftarrow t_L + \tau^M(s_M)$
$M.elapsed \leftarrow 0$

**UPON RECEIVE** $\chi$ **then**
**IF** $\chi ==$ `PAUSE`: $\mu \leftarrow PAUSED$
**IF** $\chi ==$ `RESUME`: $\mu \leftarrow RESUMED$

The time advance function $\tau$ of AS is always infinity. However, if $\Lambda \neq \emptyset$ or if an external input was received then $\tau$ evaluates to 0. The output function first sends `Y`: $(\Lambda, id, \texttt{parentId}, t_L)$ if in case $\Lambda \neq \emptyset$ and then `DONE`: $(id, \texttt{parentId}, t_N)$ or otherwise. The output is sent via the current active outport. The internal transition function $\delta_{int}$ clears $\Lambda$. The output function $\lambda(s)$ and time advance function $\tau(s)$ are extended to allow logging. Whenever any of the enumerated messages (except for $\chi$) is received, the new state of AS is output through the LOG port.

**The Coordinator model** (CO) also has one inport RECEIVE and as many SEND outports as there are machines. Its state is composed of:

– the coupled DEVS model $M$ it simulates,

– a unique identifier $id$,

– its parent identifier $parentId$,

– the last event time $t_L$,

– the next event time $t_N$,

– the list of events to be processed $L$,

– the list of children (simulators the CO coordinates down the simulators hierarchy) `children`,

– the list of children still processing an event `activeChildren`,

– the output set $\Lambda$ of `X` messages for its children,

– an output set $\Psi$ for all the events on the outports of $M$, and

– a single-rowed table `activeOutport` identifying which outport is currently active.

CO is reactive, waiting for a `*`, `X`, `Y`, or `DONE` external event as described in Algorithm 2. Note that the first two are received from its parent, while the latter two are received from its children It also receives an initialisation message `INIT` to which its sets its state. CO can also receive simulation control event $\chi$ from the CONTROL_IN port to set its mode.

**Algorithm 2** The external transition $\delta_{ext}\left((s,e),x\right)$ of a coordinator

**UPON RECEIVE** INIT:$(parent, \texttt{children'}, t'_L, t'_N, M', L', \Lambda', \Psi', \texttt{activeChildren'}, \texttt{activeOutport'})$ **then**
$M \leftarrow M'$
$parentId \leftarrow parent$
$\texttt{children} \leftarrow \texttt{children'}$
$t_L \leftarrow t'_L$
$t_N \leftarrow t'_N$
$\Lambda \leftarrow \Lambda'$
$\Psi \leftarrow \Psi'$
$\texttt{activeOutport} \leftarrow \texttt{activeOutport}''$
$\texttt{activeChildren} \leftarrow \texttt{activeChildren'}$
$\mu \leftarrow PAUSED$

**UPON RECEIVE** X:$(x, source, target, t)$ **then**
$\texttt{activeChildren} \leftarrow \texttt{activeChildren} \left\{i | self \in I_i^M\right\}$
$\Lambda \leftarrow \bigcup_{i \in \texttt{activeChildren}} Z_{self,i}^M\left(x\right)$
$t_L \leftarrow t$

**UPON RECEIVE** *:$(source, target, t)$ **then**
$\texttt{immList} \leftarrow \{i | \left(i,T\right) \in L \wedge T = t\}$
$i^* \leftarrow \texttt{select}(\texttt{immList})$
$\texttt{activeChildren} \leftarrow \texttt{activeChildren} \cup \{i^*\}$
**remove**$(i^*, L)$
$t_L \leftarrow t$

**UPON RECEIVE** Y:$(y, source, target, t)$ **then**
$\texttt{activeChildren} \leftarrow \texttt{activeChildren} \left\{i | i \in I_{source}^M \backslash \{self\}\right\}$
$\Lambda \leftarrow \bigcup_{i \in \texttt{activeChildren}} Z_{i,source}^M\left(y\right)$
$\Psi \leftarrow \Psi \cup \left\{Z_{i,self}^M\left(y\right) | i \in I_{self}^M\right\}$
$t_L \leftarrow t$

**UPON RECEIVE** DONE:$(source, target, t)$ **then**
$L \leftarrow L \tilde{\cup} \{(source, t)\}$ // replace *source* entry if it already exists in $L$, otherwise add entry to $L$
$\texttt{activeChildren} \leftarrow \texttt{activeChildren} - \{source\}$
$t_N \leftarrow \min\left(t_N, t\right)$

**UPON RECEIVE** $\chi$ **then**
**IF** $\chi ==$ PAUSE: $\mu \leftarrow PAUSED$
**IF** $\chi ==$ RESUME: $\mu \leftarrow RESUMED$

In this notation, $Z^M$ is the transducer function of $M$ and $I_i^M$ is the set of influencers of the sub-model $i$ of $M$. The time advance function $\tau$ of CO is always infinity. However, if $\Lambda \neq \emptyset$ or if $\texttt{activeChildren} = \emptyset$, then $\tau$ evaluates to 0. If $\Psi \neq \emptyset$, the output function first sends Y: $(\Psi, id, \texttt{parentId}, t)$ where $t$ is the time CO received Y. Then if $\Lambda \neq \emptyset$, X: $(\Lambda, id, \texttt{activeChildren}, t)$ is sent when Y was received or X: $(\Lambda, id, i^*, t)$ is sent when * was received. Finally, if $\texttt{activeChildren} = \emptyset$ then DONE: $(id, \texttt{parentId}, t_N)$ is sent. The internal transition function $\delta_{int}$ clears $\Lambda$ and $\Psi$. The output function $\lambda\left(s\right)$ and time advance function $\tau\left(s\right)$ are extended to allow logging.

Whenever any of the enumerated messages (except for $\chi$) is received, the new state of CO is output through the LOG port.

**The Root Coordinator model** (RC) also has one inport RECEIVE and as many SEND outports as there are machines. Its state is composed of:
- a unique identifier *id*,
- the current simulation time $T$
- the list of children (simulators at the top of the simulators hierarchy) `children`
- a termination condition $\beta$, and
- a single-rowed table `activeOutport` identifying which outport is currently active.

When a `DONE`: $(source, target, t)$ is received, $T \leftarrow t$. However if $\beta$ is not satisfied, the simulation is stopped. The time advance function $\tau$ of RC is always infinity except when `DONE` was received. The output function returns `INIT`: (*id*,`children`, 0) when the simulation starts and `*`: (*id*,`children`, $T$) when `DONE` was received.
All the simulators have a channel to each machine. This is resolved at instantiation-time. However, the simulators interact with only one machine. It is only in the case of a machine failure that the simulator will communicate with another machine.

### 4.2 Communication between simulators

Each simulation entity runs on a machine. This is modelled by a channel from the simulator to the machine being active. There can be at most one active channel at the same time.

**Local Coupling Table** (LCT) holds a table mapping each simulator running on the machine to a unique port. When it receives an event, it is forward to the appropriate port of the target after some delay time. The delay time for local search is sampled from a parameterized uniform distribution (typically order of milliseconds). However, if the target is not in the local table, it is forwarded to the REMOTE_SEND outport. LCT models intra-machine communication of simulators.

**Remote Coupling Table** (RCT) has a similar behaviour to LCT. It holds a table mapping each simulator in the cluster to the machine it is running on. The delay time is typically longer taking in consideration network communication delays (order of seconds). RCT models inter-machine communication of simulators.

**Machine Activity** Each coupled model *Machine* has a LM and an *Activity* atomic model generating failures on the machine. After some time (specified in the time advance), *Activity* sends a failure message to LCT. After some time, it can send a revival. Note that the time for revival can be infinite. When LCT receives a failure, its time advance evaluates to infinity.

### 4.3 Fault-tolerance entities

When running a distributed environment, several fault-tolerance issues must be handled. Among them is machine failures. As a consequence, mechanisms such as state restoration and resource reallocation come into place. Fig. 5 illustrates the cluster integrating all the different modelled entities.

**Monitor** server monitors each machine to detect failures. At regular time intervals, it pings all the machines through its PING outport. The monitor accumulates all responses within a certain time-out. It continues pinging (at the regular frequency) as long as it receives responses from all the machines (from its ALIVE inport). However, if timeout is reached before head, the monitor considers the remaining machines as failed. It subsequently notifies the Master.

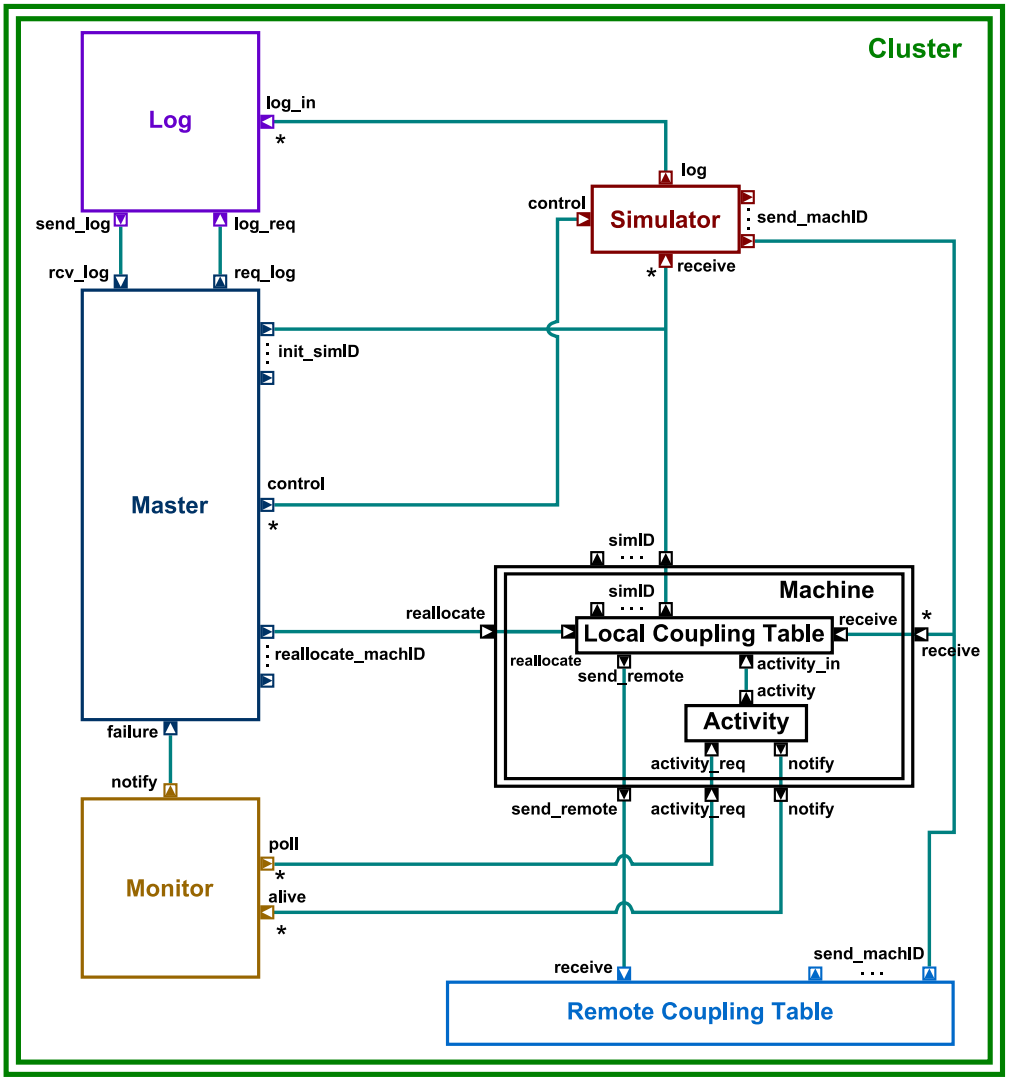


**Fig. 5.** A DEVS model representing distributed environment for DEVS simulation. A star on an inport is a shorthand notation to represent channels incoming from all the components of the same type as the source of the channel. A star on an outport is a shorthand notation to represent channels outgoing to all the components of the same type as the target of the channel. Dots between ports with a generic label is a shorthand notation to represent as many ports on the host component as there are components of the same type as the source/target of the channel. e.g., the outports labelled ports *simID* on the *Machine* component represent one outport per AS, CO, and RC.

**Log** server receives the log entries from the AS, CO, and RC entities. As described previously, the log server receives the DEVS messages sent and received, together with the simulator state and the (simulated) time stamp. A cleaning mechanism removes irrelevant traces from the log. Whenever a second entry from the same simulator is received, the log server removes the previous one. This is sufficient since state restoration is only applied from the nearest previous state.

**Master** coordinates the whole environment. Its state holds the simulation hierarchy (coupling of the AS, CO, and RC models) and the resource allocation (which simulator is currently running on which machine).
Before the simulation starts, the master sends an `INIT` message to all the simulators. The message provides knowledge of the parent of the simulator, its children (for CO and RC), the initial state the simulator will start at and the machine it will be running on. Then RO sends a `*` message to its children and the simulation runs until a termination condition is satisfied. When it receives a failure notification, the master first sends a `PAUSE` message to all the simulators to halt the simulation. We define a simulator (AS, CO, or RC) to be failed if it is allocated to a failed machine. It also request the last saved state of the simulators formerly running on the failed machines. Upon receiving these entries, the master repartitions the simulators[1] and notifies each machine is notified by its new allocation of resources. The master also sends an `INIT` message to the failed simulators. Finally it sends a `RESUME` message to all the simulators to continue simulation from their current (or newly modified) state.
We have modelled the Master, Log, and Monitor components as three different servers. It is possible to consider them as one single server. Note that the master server could even be modularly split further. This is an implementation design consideration.

### 4.4 Generic instantiation and simulation experiments

This model of a distributed DEVS simulator was implemented in `pythonDEVS`. To be able to simulate this model, several simulation experiments are provided as a library. It instantiates the Cluster *coupledDEVS* model which in turn creates the necessary AtomicSolvers and Coordinators according to the given host DEVS model (in our example, it is the client-server model that). The necessary inports, outports, and port connections are created. The parameters of the cluster are the following.

- The host model as a `pythonDEVS` data structure,
- the number of machines in the cluster,
- the initial partition of resources specifying the node location of every simulation entity,
- the initial states of all the AS, CO, and RC[2] models (a default initialisation is also provided),
- the termination condition of the simulation *i.e.*, the condition RC should wait for to end the simulation,
- the distribution of the delay for LCT to respond,
- the distribution of the delay for RCT to respond,
- the distribution of the delay for Master-Log communication (typically very fast),
- the distribution of the delay for Master-Monitor communication (typically very fast),
- the ping frequency of the Monitor, and
- the distribution of the delay for the Activity to notify that machine has not failed.

We propose an experiment where the only variable is the time before the Monitor times-out. The simulation collects performance results for different time-out values. Performance can be measured by the number of log entries in the Log server since every simulation operation of AS, CO, and RO is logged.

[1] The repartition may also need to reallocate simulators that were running on non-failed machines.
[2] the start time of the simulation can be specified in the initial state of RC.

# 5 Implementing a distributed DEVS Simulator

## 5.1 Middleware Description

**RMI/Blocking:** We have chosen the RMI (Remote Method Invocation) as a middleware layer for many reasons. It simplifies distributed computing through its ease of use for the programmers providing transparency for remote procedure calls. It also hides all internal implementations of the lower level communication to maintain remote references locally. Given the nature of our particular simulations, any type of object can be sent between the solver objects (thus a flexible solution). That would be much more tedious to implement using TCP. To accomplish this task for our python implementation of the DEVS formalism, we have chosen the PYRO framework. PYRO is an RMI-based middleware solution, written purely in python and runs in a similar fashion to JAVA RMI.

**PyRO** is short for Python Remote Objects. It is an advanced and powerful Distributed Object Technology system written entirely in Python. It abstracts away most of the network communication code. PyRO takes care of the network communication between objects once split over different machines on the network. It resembles Java's RMI. PyRO, however, has less similarity with CORBA, a system and language independent from the Distributed Object Technology.

**Factory object: distributing the different simulation solvers onto different server machines** The factory solution allows for dynamic partitioning of solver objects onto the cluster machines. Factory objects can create and host many solver objects. They represent basically a running daemon or a service which can host these solver objects. This approach will facilitate the fault tolerance aspect of the simulator which we intend to implement, namely: the re-instantiation of failed solvers throughout the simulation.

## 5.2 Distribution Issues

**Location configuration and model partitioning:** The deployment of different solver objects on different server is called partitioning. There exist several algorithms which allow for optimal partitioning, but these are not in the scope of this work. It is however possible with this implementation to allow for dynamic partitioning: meaning that an algorithm is easily pluggable into our implementation.

**Simulation Tracing - Log Server:** The simulation is meant to produce a trace which describes events occurring at certain times and how they are handled. This trace can be produced from all of the solver objects. And since in a distributed setting, these objects live on separate machine, a new mechanism to aggregate these individual traces into a single trace is needed. We have implemented this by means of a Log server. The log server can keep logs for many simulations running at the same time. It creates a simulation trace object for each simulation running, which can then be called upon to store logs from the individual solver objects. At the end of the simulation, the log server can be used to print the simulation trace of the whole run.

**Instantiation:** The initial step is to start a naming server on one of the machines on the cluster. Then a server containing a factory object, Solver Factory, is instantiated. It will subsequently register itself with the naming server. Through the naming server, the Solver Factory can be discovered by the simulation engine clients. These clients wish to instantiate solver objects, atomic or coupled, destined to live on the factory object's machine.
A remote reference is created for each factory and is used to create the solver objects. These solver objects are passed the log server reference to send their traces. We have implemented the simulator to accept a mapping object, which describes which machine in the cluster a solver should be running on. The simulator will then instantiate the solvers according to their specified location or will instantiate them locally if there is no specific mapping provided for them. Fig. 6 illustrates the overall architecture of the implementation.

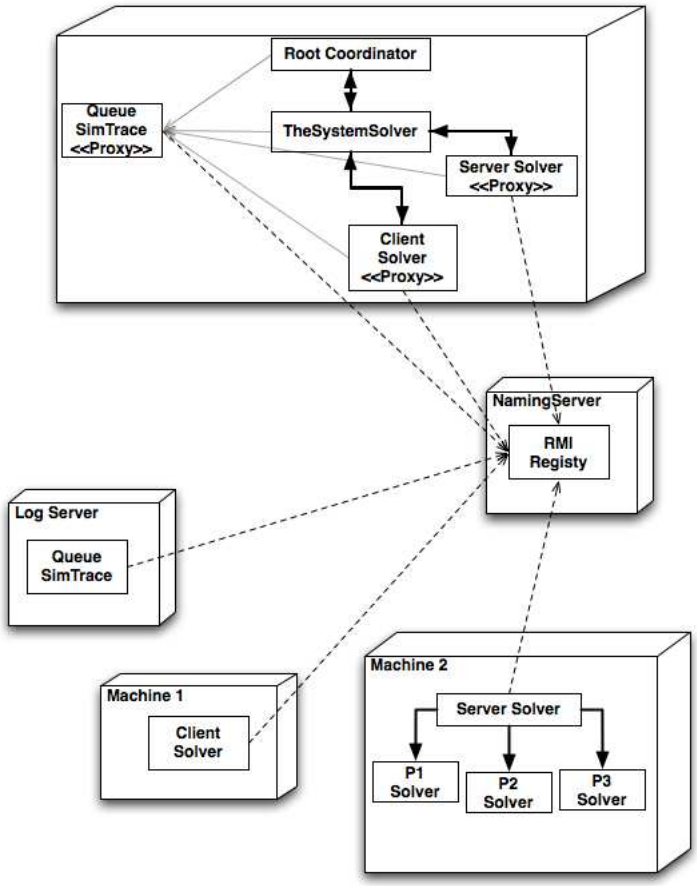


**Fig. 6.** The RMI architecture of a distributed DEVS simulator using Pyro

### 5.3 Fault tolerance

Since the solvers are dispersed over several machines, it is expected to encounter a new class of errors which were not present in the classical non-distributed DEVS simulator. Like in any distributed settings, we would have to deal with unexpected machine crashes. It is important to handle such crashes with affecting minimally the simulation flow. Assume a simulation which is supposed to run for a few days on a cluster. If one of the machines that are associated with the simulation runs into some error or if it simply crashes, the simulation will stop and would have to be manually restarted later on. This is deemed inconvenient and not scalable as we expect the possibility of machines crashing increasing with running time. As in any fault-tolerant technique, there are two phases for achieving that: fault detection and fault recovery. For fault detection, there exists many techniques, e.g., acceptance tests or $n$-version programming among others. For recovery, first perform regular backup operations whenever a modification to the state of the system (or object) takes place. Second, restore the latest fault-free state.

**Fault Model and Detection:** In the application of the distributed DEVS simulation, we will consider a fault model where cluster machines, which are part of the simulation, can crash during the simulation or even, become in a faulty state without crashing. The fault detection technique would be a timeout on methods being executed on remote objects. Meaning that if a solver failed to respond to a call (a simulation call, heartbeat call or other calls) after a certain timeout the site or the component on the site is considered crashed. The recovery technique takes over. Ideally these timeouts should be specified according to the partitioning or mapping of the simulation. This is discussed in detail in another section.

**Fault Recovery:** To recover from a fault, once detected, a strategy has to be implemented. In general, enough data needs to be collected about the state of several solvers and the DEVS models they simulate. For Atomic Solvers: $t_L$ and $t_N$. For Coordinators: $t_L$, $t_N$, *eventList*, and *subsolvers* (the solver objects, coupled or atomic, which simulate the sub-DEVS component of the current solver's DEVS model). For both solvers the DEVS model being simulated with its state.
Having this information is sufficient to restore the whole simulation at a correct state to resume the simulation. The simulation needs only to keep the last valid version of the state of all the solvers of the simulation for it to be restored. The choice of when to update this version is one of the key optimization issues. For our application we have chosen the time of this backup to be just when the Root Coordinator sends the $*$ message to its top level solver model.
The Root Coordinator holds the mapping of solvers to machines. And before sending the $*$ message, it will traverse the mapping configuration and use it to send a retrieve current state message for each solver asking it for its current state. It will then store these states for further use in case a restore is needed. Performing this operation right before sending the $*$ allows for using this stage as a check-point activity. It will be associated with the time that the $*$ message should have been sent. This is useful because, assuming that one of the solvers down the solver hierarchy have crashed after sending the $*$ message from the root coordinator. This error will be detected. The Master Server (the same as the Root Coordinator in this implementation) will be able to use on another machine the last stored and valid version of the simulation to reinitialize the solver which has crashed. It also reset the references, if any, to its corresponding parent and sub-solvers. The last thing left would be to resynchronize the states of all the solvers to the last stored valid state in the Master. Performing the backup operation could be optimized by simply pulling the changed state, rather than the new states of the solvers. The Master could be optimized to reason about which solver states it should update, due to changes by the last sent $*$ message.

### 5.4 Optimal estimation of parameters

One of the biggest configuration questions in the distributed implementation, with regards to efficient fault-tolerance, is the timeout that should be set on solver calls, before assuming that the site has crashed or even

has a fault. This timeout really depends on the model being simulated and the partitioning that is used for the solvers. The timeout value might also be different at different solver levels. Setting an arbitrary constant value may not be sufficient, as it has to take into account the network message passing time and other properties.
Network statistics for a specific cluster can be gathered using the current implementation of the log server. As discussed, the solver objects send their traces into the log server throughout the simulation. By running a simple simulation, the log server can produce some average network parameters with regards to the time a message takes to travel over the network from a solver to another in the hierarchy. These numbers can be given to the modelled distributed simulator described earlier. It then can simulate this delay which is specific to the cluster.
The trace message produced at each solver is augmented with the following parameters:

- *name* : name of the model in the solver,
- *t_global*: the global current time at which this trace was produced,
- *t_machine_receive*: the local time at the solver machine when the message was received,
- *t_machine*: the local time at the machine when the trace was produced.
  It gets augmented at the log server by:
- *t_local_received*: the local time at the machine when the trace was produced.

The optimal timeout for a specific solver message on another can be discovered using the modelled simulator: before a solver makes a call on another solver, it can invoke the modelled simulator to run a simulation of the current call and expect the time length it should take. This simulation is expected to be much faster than the actual implementation since it will only simulate and not run the actual model. The computed value is then used as a timeout limit for the call.

# 6 Current state-of-the-art of distributed DEVS

The DEVS formalism and its simulation protocol as described in section 2 is well suited for local execution on a single machine. Many approaches for distributing DEVS have been proposed and implemented in several projects (e.g., [5,6,7,4]). Thanks to the modularity and hierarchical structure of DEVS, distributing a DEVS model execution on several processes can be achieved without modifying the simulation models.
In fact the distributed architecture for DEVS can be divided into layers. The *application layer* (highest level of the system) is the modelling and simulation problem under study. The DEVS model lies in the *modelling layer*. At the *simulation layer*, the protocol to simulate the DEVS model is implemented. The lowest level layer is the *middleware layer* where the communication between computing nodes is established.
In the remainder of this report, we compare three implementations for distributing DEVS: `DEVS/Grid`, `DEVS/RMI`, and `DEVS P2P`. Note that the simulation protocols used are based on Parallel DEVS.

## 6.1 DEVS/Grid

As introduced in [5], DEVS/Grid makes use of the distributed computing paradigm: grid computing. Together with the simulator, the simulation layer comprises a model partitioner, deployer, and activator. The partitioner divides the DEVS model in partitions containing at least one DEVS block. Partitioning is computed based on the construction of a cost tree using a hierarchical reconstruction of the states of coupled models. These partitions are dispatch by the deployer to an activator on each host resolved by Grid middleware services. The message received consists of the activator's id, the blocks in the partition and the coupling information (between different partitions). The activator then creates a simulator for each DEVS block (atomic or coupled) and starts it. At that time, the necessary communication channels are automatically created based on the coupling information (how the ports are connected).
Simulators communicate at the middleware layer. A DEVS message is sent from the simulator to a proxy which encodes the message as a "Globus message". Messages are exchanged via sockets following the Globus protocol (widely used in grid computing).

The DEVS/Grid simulation protocol does not use coordinators, in contrast with the conventional abstract simulator in section 2.2. Coordinators have two roles (in parallel DEVS). One is to maintain the coupling relations between ports, but this is taken care of by the grid activators, statically and dynamically. The other role of a coordinator is to synchronize simulation time and control simulation behaviour of its (inner-)atomic simulators. For that, the DEVS/Grid protocol distinguishes protocol messages from user messages. The protocol message $TA$ is for publishing the next event time $t_N$ of a DEVS model, while *done* is used for expressing the termination of a simulator. User messages are those generated by DEVS models.
The simulation protocol works as follows. After publishing a $TA$ message, a simulator is blocked until it receives all $TA$ messages from the other simulators. It advances its simulation time by the minimum $t_N$ received. Thereafter, output is computed (if needed) and the simulator delivers the corresponding output message to the linked models via the communication layer. Upon receiving DEVS messages, each simulator publishes a *done* message and performs its delta transition function concurrently depending on the message (event) received. It then computes its new $t_N$ and the simulation cycle starts again.

### 6.2 DEVS/RMI

DEVS/RMI [7] combines `DEVSJAVA` (the implementation of a DEVS modelling and simulation framework, like `pyDEVS`) with java remote object technology JAVA/RMI.
In DEVS/RMI the DEVS model is extended with three additional key components (that can be modelled as DEVS blocks): the *configurator* for model partitioning, the *controller* for simulation control, and the *monitor* for failure detection and status verification of hosts. The configurator analyses the model received to then apply a partition/repartition algorithm. Partitioning can be done statically by assigning model partitions to remote locations during initialization phase. Moreover, dynamic partitioning is performed at run-time, passing models as parameters to remote machines. When a new partition plan needs to be computed, the controller stops the simulation until the partition information is received from the configurator. Thereafter, the controller re-configures the simulation environment and continues the simulation. Since JAVA/RMI supports persistent object migration natively, migrated or newly created simulators need not to restart the simulation from time zero and the simulation continues seamlessly. The monitor collects information about each running model in the network, measures their activities and sends that information to the configurator.
The simulation algorithm remains the same as the conventional one. However remote interfaces must be extended (using proxies) in order to communicate via JAVA/RMI and pass serialized messages. It is important to note that simulators may be created remotely or locally, depending on what is specified in the (sub-)model "where" parameter.

### 6.3 DEVS P2P

DEVS P2P allows DEVS models to be simulated on a peer-to-peer network [6]. Each DEVS block is coupled in a coupled block with "virtual" DEVS ports correctly routed according to the corresponding source/target coupling. The channels are mapped to pipes over the network.
The middleware used is JXTA. Thus a mapping from the virtual DEVS channels to JXTA pipes needs to be processed as described in [6]. DEVS P2P supports the same hierarchical partitioning and restructuring as implemented in DEVS/Grid. In DEVS P2P peers are wrappers for their local simulator (atomic or coupled). Each simulator is assigned to the active partitions of the model only, which decreases the number of simulation processes and control messages. At run-time, when a new simulator/peer is created, the new pipe is published to the network. Then, when a source pipe needs to communicate, it advertises its discovery pipe and only then the I/O communication can be established.
The peer, atomic simulator and coupled simulator protocols are described in [4]. The peer algorithm simply serves as an interface for communication between other peers over the network, wrapping message I/O handling. The atomic simulator protocol remains almost unchanged with the difference that message I/O is sent or

received via the peer it is located on. The coupled simulator protocol however, handles the coupling relation between DEVS blocks internally. The simulation time synchronization is also managed at the coupled level. The coupled simulator also processes message I/O via the peer. In the case of a remote submodel, it is the peer that communicates via the network. All messages peers exchange are serialized to JXTA messages in XML format.

When a peer delivers a $TA$ or *done* message, it is published to all connected peers. Given that it is in JXTA message format, the identifier of the peer is also passed and the JXTA layer will ensure communication with the specified peer. Reconfiguration and load balancing is managed in that layer as well. Caching is used to decrease network communication overhead.

### 6.4 Discussions

| | **DEVS/Grid** | **DEVS P2P** | **DEVS/RMI** |
|---|---|---|---|
| **Middleware** | Globus | JXTA | Neutral: RMI |
| **Portability** | No | No | Yes |
| **Dynamic reconfiguration** | Grid computation | JXTA message service | JAVA/RMI |
| **Mapping to node** | Resolved by grid | Extend each block to communicate with peer | Host IP passed as parameter for each block |
| **Partitioning** | GMP | AHMP | Any |
| **Message exchange** | GIIS | JXTA message format | Serializable object |
| **Simulation protocol** | Explicit message delivery | Modified to communicate through peer | Unchanged, interfacing with RMI technology |

**Table 1.** Matrix comparison

Distributed DEVS is achieved in layers. As shown in the previous section, the three approaches differ at the *middleware layer*, but also in the *simulation layer*.

DEVS P2P strongly relies on JXTA Pipe Interface as middleware to support distributed execution of DEVS. DEVS/Grid relies on a grid infrastructure "Globus" for its communication layer. The most neutral approach is DEVS/RMI which uses the RMI technology (in this case Java/RMI). The advantage of using RMI is the portability of the distributed simulation framework. Because RMI supports persistent data migration on remote machines, DEVS/RMI natively supports dynamic model structure change. In DEVS/Grid, the grid layer requires explicit specification when migrating the state of a simulator. As for DEVS P2P, this is handled through JXTA services. Additionally, each block is coupled with a "virtual" block to communicate between peers. Similarly, Grid resolves simulator mapping on computing nodes by the activator. In DEVS/RMI, the location of the simulator is specified statically as a parameter in the block. This can then be changed by the controller at run-time. As far as model partitioning is concerned, DEVS/Grid uses the Generic Model Partitioning Algorithm (GMP) to a cost tree that it constructs. DEVS P2P uses a more cost-efficient algorithm, the Autonomous Hierarchical Model Partitioner (AHMP). DEVS P2P and DEVS/Grid use a message exchange format respective to their middleware constraint, while RMI supports serialized objects to be exchanged. DEVS P2P simulation protocol forces all none I/O messages are exchanged locally between the simulator and its peer. In the DEVS/Grid environment, the simulation protocol must explicitly deal with delivering messages to remote simulators. As for DEVS/RMI, the simulation protocol must implement the RMI interface, without changing the conventional protocol. DEVS P2P and DEVS/Grid need an additional layer for simulation time management. These comparisons are summarized in Table 1.

# 7 Conclusion

In this report we have described how a distributed DEVS simulator can be modelled in classical DEVS. This model allows us to simulate and try out our fault-tolerance solutions for different scenarios such as machine failures. The simulated model was then implemented as a distributed simulator using an RMI technology. Trace matching was used to ensure consistency between the two approaches and the original non-distributed simulator. Moreover, the implementation is used to collect statistics about the network by running some sample experiments. These statistics, such as the message passing and response times of different servers and solver objects, is given back to the simulator model. The simulator model can then use these values to simulate network delays. Consequently, it can expect more accurately optimal timeout intervals throughout the simulation. Furthermore, the implemented distributed DEVS simulator can launch simulation experiments to determine optimal values of its parameters. At run time, the executions of such experiments should be very fast compared to the actual execution since they only simulate the network delays and not execute the model.
We proposed to compute the optimal time-out duration before resource reallocation. This approach of simulating "on the side" different possible scenarios given the current state while running a DEVS simulation in a distributed environment may solve important issues in such distributed systems. Once the model has the information it needs to simulate the network delays in a given cluster, it could be then used to simulate different machine-resource repartitioning possibilities and find the optimal configuration. Fault-tolerance and load balancing can be fine-grained with this approach.
Another possible future work would be to generate the implementation code form the simulator model, by passing just few parameters which are needed. This will allow for simulating complex experiments in a distributed setting in a more automated way. In fact, this would make a perfect fit for a distributed DEVS simulation framework to be used on any cluster.

# References


1. B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.
2. J.-S. Bolduc and H. Vangheluwe, "The modelling and simulation package pythonDEVS for classical hierarchical DEVS," McGill University, MSDL Technical Report MSDL-TR-2001–01, June 2001.
3. A. C.-H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism," *TSCS*, vol. 13, pp. 55–67, 1996.
4. P. Sunwoo, S. H. J. Kim, C. A. Hunt, and D. Park, "DEVS peer-to-peer protocol for distributed and parallel simulation of hierarchical and decomposable DEVS models," in *ISITC'07*, H. Kim and B. Wah, Eds. Jeonju(Korea): IEEE Computer Society, November 2007, pp. 91–95.
5. C. Seo, S. Park, K. Byounguk, S. Cheon, and B. P. Zeigler, "Implementation of distributed high-performance DEVS simulation framework in the grid computing environment," in *ASTC'04*, H. Unger, Ed. Arlington (USA): Society for Modeling and Simulation International, April 2004.
6. S. Cheon, C. Seo, S. Park, and B. P. Zeigler, "Design and implementation of distributed DEVS simulation in a peer to peer network system," in *ASTC'04*, H. Unger, Ed. Arlington (USA): Society for Modeling and Simulation International, April 2004, pp. 18–22.
7. M. Zhang, B. P. Zeigler, and P. Hammonds, "DEVS/RMI-an auto-adaptive and reconfigurable distributed simulation environment for engineering studies," *Journal of Test and Evaluation*, vol. 27, no. 1, pp. 49–60, April 2006.