Alcom-FT Technical Report Series
ALCOMFT-TR-03-155

DAP: A Generic Platform for the Simulation of Distributed Algorithms

Ioannis Chatzigiannakis* Athanasios Kinalis* Athanasios Poulakidas*
Grigorios Prasinos* Christos Zaroliagis*

December 1, 2003

Abstract

DAP (Distributed Algorithms Platform) is a generic and homogeneous simulation environment aiming at the implementation, simulation, and testing of distributed algorithms for wired and wireless networks. In this work, we present its architecture, the most important design decisions, and discuss its distinct features and functionalities. DAP allows the algorithm designer to implement a distributed protocol by creating his own customized environment, and programming in a standard programming language in a style very similar to that of a real-world application. DAP provides a graphical user interface that allows the designer to monitor and control the execution of simulations, visualize algorithms, as well as gather statistics and other information for their experimental analysis and testing, respectively.

1 Introduction

Distributed systems are today ubiquitous in many aspects of social life, in business, academia and home. It seems that this trend will continue and we will witness the emergence of distributed applications with increased power and complexity.

The design of distributed systems presents many challenges both at the theoretical and the implementation level [23]. At the theoretical level the designer has to tackle the fundamental problems of asynchrony (events take place at times that cannot be known in advance), limited local knowledge (each computing entity is only aware of information that it acquires and has no global view of the whole environment), and failures. At the implementation level the difficulties stem from the fact that it is almost impossible to achieve scalability, that is, to create a testbed that resembles the enviroment where the application is going to be deployed (e.g., a network of hundreds of computers). This critical issue of scalability can only be addressed via simulations. Simulation environments play a crucial role in the development of distributed algorithms and their practical assessment, provided that they model accurately the application environment and they are easily usable by the end-user. However, it is not at all obvious how to achieve these goals.

Simulation environments that provide all the necessary primitives to allow simulation of any distributed algorithm are not so many. To the best of our knowledge there are two such environments: DSP [4], and IOA [12]. On the other hand, there are several environments for the simulation of

*Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece, and Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece. Emails: {ichatz,kinalis,green,zaro}@ceid.upatras.gr, poulak@cti.gr. This work was partially supported by the IST Programme of EU under contract no. IST-1999-14186 (ALCOM-FT).

network algorithms (i.e., distributed protocols with low-level functionality), or environments that focus on the simulation of a specific area of research in distributed computing. Important examples of such environments include ns [19], YACSIM [15], and SimUTC [24]. Another crucial characteristic is that some simulation environments request that a user develops a protocol using a specific description language provided by the environment, or a scripting language (this is the case for ns, IOA, DSP). This should be contrasted to simulation environments (e.g., YACSIM, SimUTC) that allow users to develop programs in a standard programming language (`C` or `C++`), which could be advantageous in the sense that the same program can be directly run on a real distributed environment. We give a brief overview of the above mentioned environments.

The network simulator, `ns`, [19] is a discrete event simulator aiming at simulating network protocols for low-level functionality, that is, simulation of TCP-like protocols, as well as of routing and multicast protocols over wired and wireless (local and satellite) networks. `ns` requires that the protocols and the simulation setup is written in the OTcl scripting language (Tcl with object-oriented extensions by MIT). A user may develop protocols in a standard programming language (`C++`), but needs to bind them to OTcl in order to be simulated by ns. An extended library of network-level services has gradually been built for ns, including an animation tool (nam) for animating the simulation.

The IOA project [12] provides a formal language for describing processes that are modeled using I/O automata. To model a distributed algorithm or system, a user has to program in the IOA language. A set of tools for this language (including a simulator) that will provide support for the development, analysis, and simulation of IOA programs, is under current research. The set of tools is developed in `Java`. Using a tailored language like IOA, or Esterel [6] (a synchronous language for programming reactive systems, e.g., real-time systems and control automata), may be more convenient in some applications, but it has a portability penalty (efficient compilers may not be available on some platforms) and a learning curve.

The Distributed Systems Platform (DSP) [4] is a software platform designed for the implementation, simulation, and testing of distributed algorithms, and it offers a set of tools which allow the researcher and the algorithm designer to work under a familiar graphical and algorithmic environment. On the other hand, DSP provides its own set of simple, algorithmic languages (DSPL) which can describe the topology and the behavior of distributed systems and it can support the testing process (on-line simulation management, selective tracing, and presentation of results) during the execution of specific and complex simulation scenarios. DSP has been implemented in `C`. It uses a client/server architecture where the client edits and sends the necessary information (the description of the protocol, the topology, the simulation initializations and actions, like when some failure should happen) to the server. The server compiles an executable. In doing that it may also link some protocol from its database that the client requested. The executable runs at the server and its simulation is transmitted and viewed at the client. This scheme is particularly advantageous for teaching.

The advantages of extending a general programming language by adding simulation routines has been noticed before. A specific example is YACSIM [15], a discrete event simulator implemented in `C` that provides a basic set of `C` subroutines that the developer can link with his program in order to produce an executable that performs the simulation (i.e., YACSIM does not provide a separate simulator, but the simulator is contained in the executable produced by the user). YACSIM extends `C` by adding simulation objects, which are data structures and a set of operations on them. These objects represent activities, queues or statistical records. The activities are either events (for event-driven simulation), or processes (for process-oriented simulation). The queues are used to delay the occurrence of an event or the progress of a process until some conditions are met. In order to perform a simulation the user has to manipulate YACSIM's simulation objects in his code and has to compile everything into a single executable. YACSIM is normally used as a base to produce more sophisticated simulators. For example, it was used to produce NETSIM [14], a general purpose interconnection network simulator for parallel architectures.

SimUTC [24] is a simulation toolkit intended for the specific problem of clock synchronization. It is round-based and built on `C++SIM` [17]. The latter is a toolkit written in `C++` that implements facilities provided in SIMULA [5]. It uses threads and is a process-oriented continuous-time discrete-event simulator. It is interesting that the `C++SIM` developers observe in [17] that `C++` compilers produce much more efficient code than SIMULA, thus resulting in faster simulations.

In this paper we present DAP (Distributed Algorithms Platform), a homogeneous environment, for the implementation, simulation, and testing of any distributed algorithm. DAP focuses on implementing distributed algorithmic ideas developed either for wired or for mobile wireless network systems. In developing DAP, we set up the following goals that actually determine its distinct features:

- To allow the client programmer to create his own environment (abstracting away irrelevant layers, such as the physical layer) with sufficient flexibility to model diverse situations (e.g. wired or wireless mobile networks).
- To allow the client programmer to program in a natural style, very similar to the style of programming for the final real-world application, using a well-known language.
- To be of substantial help to the algorithm designer by providing him with the means to monitor the execution of the algorithm and to extract detailed statistical information.
- To be used both as a tool for experimental analysis and for demonstration or educational purposes.
- To be easy to extend or integrate in other systems.
- To be efficient and easy to use.

Let us now discuss why these issues are important and how DAP addresses them.

Many systems (e.g., `ns`) are too generic and incorporate much more detail than it is needed for distributed algorithm simulation. In DAP all the less relevant lower levels are abstracted in a Topology layer. In such a simple but powerful model it is easier to capture some important aspects of a distributed environment such as asynchrony and failures of computing entities or communication channels. The user is free to customize every aspect of the simulation environment according to his own needs by setting the properties of every simulation object to constant or *variable* values (by selecting a random variable implementing some probability distribution). Furthermore, the user may define *scenarios*, which are series of actions (like node failures, interrupts, etc.) to be executed at specific points (also random variables, if needed) during the simulation, thus allowing vigorous testing of particular situations. In addition to the simplicity of the model, DAP provides a graphical topology and scenario editor for constructing the required environment.

Another disadvantage of the majority of the available systems is that they require the developer to code his algorithm in a language and a style that are specific for the system. In this case, the programmer is faced with a steep learning curve and there is a lot of wasted effort until he is able to utilize the full potential of the system. If, in addition, the language is inefficient (e.g., an interpreted language) the development/testing cycle becomes too long and the developer cannot assess easily the quality of his algorithm. In DAP the developer can program in a natural style (as in any other program) using `C++`. `C++` is a very efficient language, with a vast array of available tools. It is suited for simulations and is a language most developers feel comfortable with. Furthermore, there is another interesting perspective: after an algorithm has been evaluated using DAP, the same code can be deployed on a *real* distributed environment, without serious modifications and without the use of the simulator, just by linking with libraries we can provide.

The whole point of a simulation is to assess the quality of an algorithm. For this purpose DAP includes a graphical user interface (GUI) that can be used to monitor most aspects of the simulation in *real time*. This feature is quite important since it allows the developer to pause the simulation, make changes to the simulation environment and continue the simulation in order to measure the effect of certain events on the robustness of the algorithm. The GUI includes modules for designing the topology, constructing scenarios, recording the simulation and viewing statistics. A powerful feature of DAP is that the *same* simulation can be observed by *more than one* user interfaces (but controlled only by one) on different machines making it an ideal platform for demonstration or educational purposes. In addition to the statistics shown by the GUI, the developer is able to measure any parameters he is interested in through simple method calls. Both the statistics gathered by the user and the statitics gathered by the system can be presented in various ways (as graphs) or exported (in standard format) for use by other programs.

DAP has a flexible modular architecture and uses an open format (encoded in simple XML) for representing and exchanging information (e.g., topology, statistics, communication between modules, etc.). These features result in a variety of interesting possibilities, giving DAP some unique characteristics. The capability of having more than one users from different computers monitoring the same simulation is one of them. A more interesting one is that the simulation itself can be distributed: the simulation of some computing entities may be run on different computers (that are of course connected to the computer running the simulation core). Thus the simulator can take better advantage of available computing resources. The modularity of the system is such that a developer can even code his algorithm in *any* language of his choice by following our communication protocol (our library, of course, abstracts this communication and provides `C++` bindings directly). Another more practical feature made possible by the use of standard open formats is that statistical data can be exported to different programs for further processing (e.g., statistical packages, spreadsheets, etc.).

An important consideration is the efficiency of the simulation environment. In DAP this is achieved by the use of `C++`, and a straightforward programming model (i.e., there is not any "intermediate" state, or any interpreter involved).

The use of DAP is rather intuitive and does not distract the developer from his main task, that is, the design, development and testing of his algorithm. The DAP platform along with further supporting documentation is available at `http://ru1.cti.gr/~LEP-DAP/`.

The rest of this paper is organized as follows. In Section 2, we present the high-level design of DAP. In Section 3 we discuss the design of the Simulator Engine which is the heart of the simulator. The library that provides the interface between an algorithm and the simulator core is described in Section 4. Section 5 discusses the graphical user interface which is the prime medium between the user and the system. The communication protocols of DAP and the features that enable interoperability with ohter applications are presented in Section 6, while in Section 7 we explain how an interested user can develop an algorithm using DAP. We conclude in Section 8.

2 The Architecture of DAP at High-level

The main difference between DAP and similar systems is that every simulated process is actually run by DAP as a different "real" (in the operating system sense) process. This decision adds some amount of complexity to the system but it has also four, very important benefits:

1. The developer can code each process without having to follow any specific programming model (e.g., event-driven).

2. The execution of the simulation is asynchronous, by the nature of how operating systems manage execution.

3. The processes need not be executing all on the same machine. On the contrary, the simulation itself may be distributed.

4. A process may block without affecting the others. This observation is important: under this model it is easy to implement the *blocking receive* directive, a common challenge for distributed simulators.

On the basis of this design choice, DAP consists mainly of three components (see Fig. 1).

The *simulator engine* is the component that actually simulates the distributed system. It handles the set-up of the simulation, the processing of simulation events, the gathering of statistics, the communication with the graphical user interface, etc. We will examine it in detail in the next section.

The *Graphical User Interface (GUI)* is used by the user to set initialization parameters, to control the simulation and generally to monitor its execution. The GUI is a separate executable and communicates with the engine through normal TCP/IP sockets. This fact gives DAP a lot of flexibility. The use of GUI is not strictly necessary as the simulation can be set up and started by using appropriate command-line options on the main simulation engine. Furthermore, the GUI can disconnect and reconnect at any time, so the user can set-up a lengthy simulation, start it up, disconnect and then connect only at regular intervals to monitor the execution. Since the engine and the GUI communicate over TCP/IP it is possible that the simulation runs on a fast computing server and the GUI on a less powerful computer (e.g., the developer's laptop). The engine supports also the connection (at any time) of more than one GUI programs. In this case one is designated as a *controller* and is used to control the simulation, while the others are passive *observers*. The restriction of only one controller is not a technical limitation, but just a feature in order to allow centralized control which is typically required; for example, in classes where students observe the simulations controlled by the instructor. Although we currently provide a GUI written in `C++` (with the use of LEDA [20]), it is possible to write observers in any language by following our simple communications protocol.

The third component is the *DAP library* (usually abbreviated to *libDAP*). The library is linked to every executable that a developer produces and allows it to communicate with the simulator engine. The DAP library implements all the directives that are needed for the implementation of a distributed algorithm (e.g., send/receive).

All the modules are written in `C++` with some help from LEDA, the library of efficient data types and algorithms [20]. A user of DAP develops the code for his distributed algorithm in `C++` (possibly using data types and algorithms from LEDA), compiles it using a `C++` compiler, and links it with the DAP librariy to produce an executable. The user can produce more executables exactly in the same way, for the case where more than one algorithms are needed.

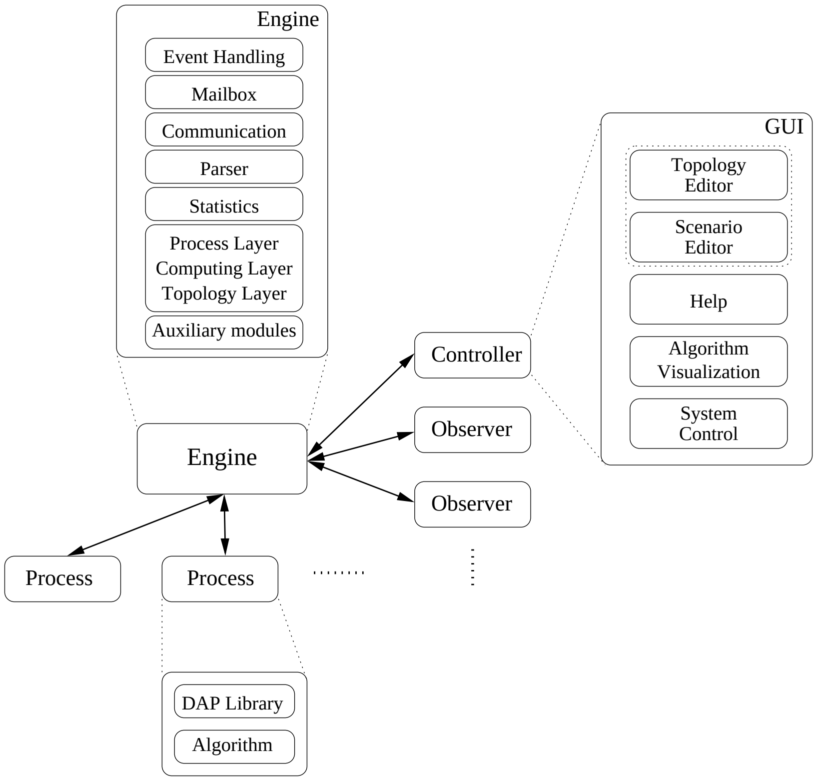

Figure 1: High-level architecture diagram for DAP.

3 The Simulator Engine

The *Simulator Engine* (henceforth abbreviated to *engine*) is the component of DAP which performs the simulation of the distributed system. The characteristics of the simulation system – most of them defined through the GUI – are stored here. The engine performs the simulation interacting with the GUI and the executables. It updates the visualization data for the GUI and changes the characteristics of the simulation if the user specifies so through the Controller.

3.1 Simulation model

The engine operates on a discrete simulation model, that is, a model where the system can change its state at only a countable number of points in time. The simulation is discrete-event [16].

The central component of the Engine is an *Event queue*. Most actions on the system (e.g., the transmission of a message) generate an appropriate event which is inserted in the event queue along with a timestamp that specifies the time of its execution.

The Engine receives communication from some simulated process, or from the GUI. In the former case, it may be a simulated message transmission, coordinating communication with a simulated process, or output from a simulated process intended for the GUI. In the latter case, where the communication comes from the GUI (in particular, the Controller), it may be events that should be inserted in the event queue (like node/channel should go up/down), or instructions to control the simulation run (like start, pause, resume, step, set delay). Each event is modeled by a class that includes a handling method that specifies its execution logic. The simulator engine gets the next event to be simulated, updates a *Global Clock* accordingly and gives control to the handling method, passing it as an argument a class that encapsulates the characteristics of the simulated system. The handling method may change the characteristics of a system (e.g., a node failure event) or generate more events (e.g., a "send message" event generates "receive message" events with the appropriate delay).

The *Global Clock* indicates the current simulation time, say t. All events that are specified to happen at time t need to be handled. In particular, the Global Clock is increased from the current value t to the value t' when all of the following conditions are satisfied:

- All events scheduled for time t have been handled.
- t' is the value of the earliest event in the Event Queue that has not been handled yet.
- No simulated process can generate an event scheduled for some time before t'. This is true if each simulated process has reached time t', has completed, or is blocked waiting for some event (like a message receipt).

3.2 Sources of input

The Engine receives input from many sources: the running processes (requesting for example to send a message), the controller, the observers, or from appropriate local files when running without the GUI.

3.2.1 Scenarios

The engine has the functionality to support user-defined scenarios, that is, a series of (simulation) actions (like node failures, interrupts, etc.) to be executed at a specified point in time during the simulation. The main use of scenarios is to allow repetitive of particular situations. It is technically possible that actions are sent to the engine at any moment (if, of course, their intended execution time is later than the current time). Indeed, since some actions can also be triggered by user actions on the GUI (like node failures), actions related to scenarios use the same protocol as the ones coming from the GUI. However, the user may also group a series of actions into a scenario and save them to a file that can be loaded by the engine. Currently the engine supports the scenario actions provided in the following list, but more can be easily added.

- Boot/start a node/process at a specified time; if no such action is specified for a node/process, it starts by default at time 0.
- Create interrupt on a node/process at a specified time; optionally repeat every some time. These interrupts are handled by interrupt functions that the user registers with `libDAP`.
- Schedule a node or channel to fail permanently/temporary at a specified time. The user can specify also the probability that the event occurs, its duration[1], or the repeat interval.
- Schedule a node to recover (with a specified probability); can be used to recover from permanent failures.

In all the actions, time, interval, duration and probability values can be specified to be random.

[1] Only valid for temporary failures.

3.2.2 Handling

Since input comes from many different sources, and the input may trigger a variety of situations (from generating an event to stopping the simulation), its handling becomes a complex task. In order to manage the complexity we use the *command* design pattern [8]. All the possible actions are represented by command classes which encapsulate their processing logic. When input arrives, a *parser* constructs the appropriate command and sends it back to the engine core. The engine then executes the command, passing it as an argument a class that encapsulates the characteristics of the system. Under this framework it is a simple matter to add new features and capabilities to our system.

3.3 Modeling of the simulated environment

The most important feature of the engine is the way the simulated environment is modeled. Instead of representing the full detail of the environment (a situation that would place an unnecessary burden on the developer), its whole complexity is abstracted in three layers:

Topology Layer. It describes the location properties of the computing nodes. It includes *topology nodes* that specify positions and *topology links* that specify connections between these positions.

Computing Layer. It describes the properties of the computing nodes. Each computing node resides on a topology node.

Process Layer. It represents the processes that make up the simulated algorithm. Each process executes on a computing node. The processes may modify characteristics of the lower layers (e.g., close a communications channel or move the computing node from one topology node to another).

This model is simple yet powerful and can capture the elements of many environments. The exact meaning of each layer depends on the application domain that is being simulated. For example, the links of the topology layer may correspond to physical connections (e.g., cables) or to virtual channels (e.g., the transmission range of a mobile phone). Every object on this model has some associated properties (e.g., channel bandwidth). Each property value may be constant or random (drawing numbers from common random distributions). DAP provides default configurations for these properties, but the user is free to provide his own values and even his own properties. This way the model can be customised to account for any distributed environment. These properties as well as the whole configuration of the environment may be static or may change over time. As an example, nodes and links may fail, or computing nodes may move to different locations.

DAP focuses mainly on algorithms for wired or mobile wireless networks. On the wired case, topology nodes and channels specify the actual configuration of the network, i.e. the locations of the computers and the connections between them. The computing nodes describe the properties of the computers (e.g., cpu speed). The process layer represents the actual processes running on the computers.

For the mobile wireless case, DAP utilizes the *motion graph model* presented in [10] to abstract the environment where the stations move. The three-dimensional space is abstracted by a *motion graph* (i.e., the detailed geometric characteristics of the motion are neglected). This model quantizes the motion space into cubes. Each cube has a volume that approximates (from below) the volume of a sphere which represents the transmission range of a mobile host. The motion graph has a vertex for each cube of the quantization. Two vertices are connected by an edge if their corresponding cubes are adjacent. In our three-layer model the motion graph corresponds to the Topology layer with each cube being a node and each edge a link. The computing layer describes the mobile nodes (also called *mobile hosts*) of the system (e.g., mobile phones) and the process layer the different programs that may run inside the mobile nodes (e.g., the program handling incoming calls).

Especially for the mobile wireless domain DAP offers extra functionality about the movement of the computing nodes. The developer may decide to use his own methods for handling movement or let the engine move the nodes according to a set of commonly used *motion patterns*. DAP provides three motion patterns.

Random Waypoint: In this model [1, 13], the mobile host selects a destination randomly within the area of motion and moves towards that location using the shortest path. The mobile host stays at the new position for some time and then starts over.

Random Walk: In this model [2, 10, 11], the mobile host selects randomly one of the outgoing links adjacent to its current position and moves to the location at the opposite end of the link. The mobile host stays at the new position for some time and then starts over.

Arbitrary Motion: In this model [3], the developer gives a series of positions and delays. The mobile host stays at each of the positions for the specified delay. After the last position the mobile host follows the list backwards, and then it starts over.

3.4 Auxiliary modules

In addition to the above, there are a few modules for a manifold of small but important tasks.

3.4.1 Communications

Communication between the engine and the rest of the system (processes and controller/observers) is done through the *tranceiver* module. This module sets up and maintains a mapping between the process IDs, the controller and the observers on the one side with their corresponding network (TCP/IP) addresses on the other. This way the engine core does not need to have any knowledge of the network details.

3.4.2 Statistics

We distinguish between *hard-wired* and *user-defined* statistics. The former refers to the statistics that are built-in the platform and are provided to any algorithm implemented in DAP. The latter refers to a mechanism that allows the user to easily define and display his own (additional) statistics.

Hard-wired statistics refer to the most commonly requested statistics from a distributed algorithm. Assume that each message is assigned a tag denoting its type. Then, hard-wired statistics refer to the number of messages sent per tag, number of messages received per tag, number of messages lost per tag, number of node failures and recoveries, number of channel failures and recoveries, average delay of messages per tag, etc.

Depending on the particular algorithm being implemented, the user may wish to gather his own specific statistics (e.g., memory usage per node of the network). To gather user-defined statistcs, the following libDAP function is provided which can be called by any process at any time t:

```
void plot(string windowID, double yValue, bool total)
```

This function allows displaying on the window specified by `windowID` a two-dimensional plot of points (`yValue`, t). The parameter `total` takes boolean values. True denotes that we want this statistic cumulatively for all processes, while false denotes that we want this statistic for a specific process (the one which calls the function `plot`).

A special module handles the gathering of these statistics. This module is available to all the event handling routines, so that every event can record appropriate details.

3.4.3 Mailbox

This module handles the storage and transmission of messages. Messages arriving for a process are first buffered here and are forwarded to the process one at a time when a process issues a message request directive. This strategy allows DAP to be independent of the underlying transmission mechanism and to have greater control on the implementation of the "blocking receive" directive. The processes are unaware of the buffering so that the message handling on their side can be kept very simple. The mailbox understands, and handles with priority, messages related to simulation flow of control (e.g., a quit message).

3.4.4 Timekeeper

Timekeeper keeps the estimate of the engine about the clocks of each process. This information, and especially the smallest clock value, is used by the engine core to coordinate the simulation. A data structure that provides efficient extraction of the minimum value along with frequent value updates is needed in this module.

3.4.5 Recorder

It is possible to record the whole simulation run and replay it later by running only the engine. The *Recorder* module is used to store all the needed information. In addition to be replayed by the engine, the recorded information can be stored and used for further analysis.

3.4.6 Logger

The Logger is a low-level module that stores detailed information about the simulation run. The module is available to all the other components. Even the processes may make an entry (through the engine) by issuing a specific directive. The level of detail can be customized and the output can be used for debugging or detailed analysis.

4 The DAP library – libDAP

The DAP library lies between the code developed by a user and the simulator. It provides to the processes written by the user access to certain *services* (or *directives*) available to a distributed system. These services are, of course, simulated by the DAP simulator. Our aim was to provide a set of services powerful enough to represent most distributed systems of interest, yet easy to learn and use (see Section 7 for an example of using the library). If needed, we can easily extend libDAP to support other functionalities that are useful in our setting (taking e.g., ideas from MPI [21] and PVM [22]).

Current services include finding the ID of a process, the computing and the topology node it resides on, the adjacent channels, its neighbors in adjacent nodes or other processes in the same node and querying the properties of these entities. A process may send a tagged message to a particular other process, to all processes at an adjacent node, to all neighbors, or to a particular *port*[2]. To receive messages, blocking and non-blocking mechanisms are supported. Depending on the application domain, there are also directives for moving the process or the computing node, or adding new links.

A process may also change the color of a node or a channel, or the label of the current node, providing simple user-control mechanisms to enhance the visualization of the simulation run. The input/output of a process is performed throught the GUI.

A timeout mechanism is supported, allowing to schedule a wake-up call. A process may register its own functions to handle such calls or other interrupts.

The DAP library is modular, and the user may specify which directives he needs in his application domain. The modularity of the library combined with the modularity of the engine and the simple communication protocol make it straightforward to extend the system with more functionality.

5 The Graphical User Interface

The DAP Graphical User Interface (GUI) provides two predominant ways of presenting information to the user in order to assist him in understanding and analyzing complex executions, and simultaneously monitor the state of a potentially large number of nodes. The information is presented to the user using both textual techniques like event and message traces and also with graphical means, by using LEDA's windows data type that provides an interface for the graphical input and output for both the X11 system on Unix platforms and for Microsoft Windows systems [20, Ch. 11].

The GUI comes in two flavors. The strong (complete) flavor is called the *Controller* and includes the following modules (see Fig. 1 and 3):

- A *Topology and Scenario editor* module which can be used to define the topology (connectivity graph) of a wired or a fixed wireless distributed system, to define the connectivity graph as well as the graph describing the space of motions of mobile hosts in a mobile wireless distributed system and to define simulation scenarios (see Section 3.2.1).
- An *Algorithm Visualization* module that allows the user to view the execution of the simulation.
- An *Event & Log Viewer* component that displays the messages generated by the simulator engine (and is driven by the Algorithm Visualization module).
- A *System Control* module that allows the user to control the other GUI modules and the simulation (e.g., increase the speed, pause it, change the state of a link to failed, etc).
- A *Help* module which provides online help.

The weaker flavor of GUI is called the *Observer*. An Observer includes only the Help and the Algorithm Visualization modules. Many Observers may be connected to the engine and view the

[2]Ports are defined as part of the topology of a network and are used to specify direction, like left-right or north-east-west-south, etc.

execution of a simulation at any time. In contrast, only one Controller is allowed to connect to the engine and control the simulation. This scheme allows the use of DAP as demostration platform.

The various components of GUI perform necessary operations in a user-friendly manner and greatly enhance the usability of DAP. The Topology and Scenario editor supplies the user with the means to customize every aspect of the simulation enviroment. Furthermore the topology and scenarios created can be saved in XML format (see Section 6) and reloaded later thus allowing the reusability of topologies and scenarios. The System Control module gives to the user fine-grained control of the simulation flow and the ability to act upon the simulation objects by temporarily suspending the execution (pause) and performing various operations, like changing the status of a mobile host (e.g., switching it off, or restarting it) or causing links to fail. The Algorithm Visualization module displays a detailed view of the current simulation enviroment and depicts the state of all simulated objects (see Fig. 2). Using these tools, the user can easily setup a simulation enviroment and control the simulation without knowledge of the underlying details.

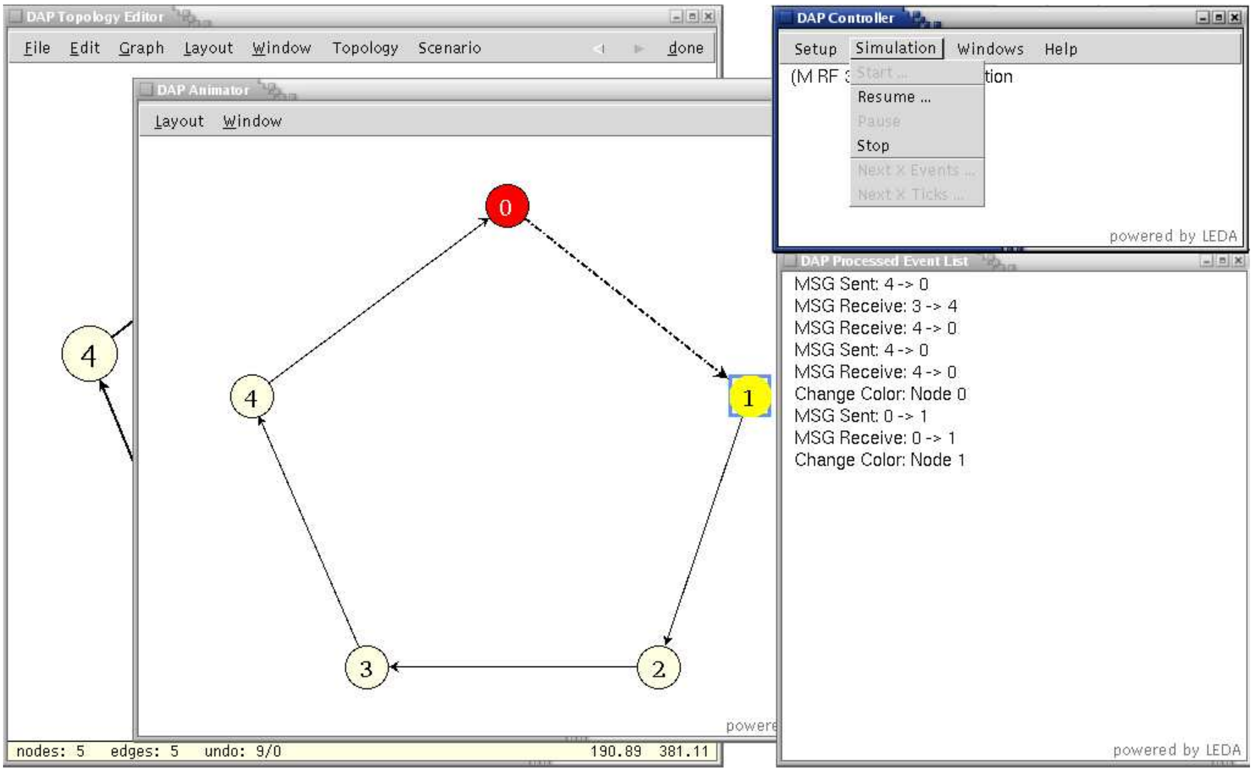

Figure 2: GUI windows while executing the algorithm given in Fig. 4. The windows shown are the algorithm visualization module in the center, the event viewer on the right, the controller on the top-right and the topology editor in the background. The elected node is the red (shaded) node in the upper area of the algorithm visualization window. Node 1, enclosed in a box, has just received the notification message through channel (0,1), which appears dashed, and has changed its color. The user has paused the simulation.

The GUI is implemented in C++, using LEDA's windowing system for the graphical part of the system. Apart from the graphical modules, there exist other modules that implement the required functionality and form a complex but efficient architecture.

The core of the GUI architecture consists of three components: the Topology manager, the

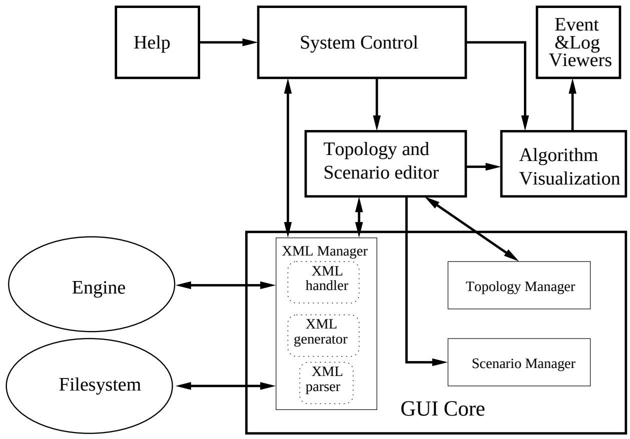

Figure 3: Major modules of the GUI architecture.

Scenario manager and the XML manager. The Topology manager holds all information related to the topology, computing and process layers, and exposes this information to the other modules through simple function calls. Furthermore, the Topology manager can export and import all topology information to and from the XML representation, with the help of the XML manager. The Scenario manager is similar to the Topology manager but instead of the topology it holds all actions related to the simulation objects and is responsible for the XML representation of scenarios. The XML manager is divided into submodules which implement functionality for the generation, parsing and handling of information represented in the XML format. It is one of the most important modules of the GUI since all data exchange between the GUI and the simulation engine is done using an XML protocol (see Section 6).

On top of the core lie the graphical modules, with the most important being the System Control module. The System Control module is responsible for the invocation and coordination of the other graphical modules as well as the control of the simulation engine and the dispatch of simulation events to the appriopriate components. The System Control module relies heavily on the XML manager for the communication with the engine. The Topology and Scenario editor module is tightly coupled to the Topology and Scenario managers and provides an association betweent the topology information and the LEDA graph data type allowing the powerful graph editing capabilities of LEDA to be used. It provides additionally a graphical interface to edit the information that can't be handled by LEDA for the rest of the topology and scenario objects. The Algorithm Visualization module is invoked by the System Control module when the simulation begins. It acquires the visual information from the Topology editor module and visualizes the topology and the execution of the algorithm as messages from the simulation engine are received. It is also responsible for the logging of events. Figure 3 depicts the GUI architecture with arrows representing the direction of data flow.

6 Communication protocols and interoperability

Since DAP is separated into many components there is a need for a mechanism that all components can use to interact with each other. The open-ended architecture of DAP makes it possible to combine DAP-supplied modules with modules supplied from third-parties. In order to ensure that third-party developers can produce new modules with ease, DAP uses simple and open communication protocols.

These protocols are formulated in the XML description language, a specification developed by the World Wide Web Consortium [25]. To use the terms defined for XML the intermodule communication of DAP is an "application" of XML. XML is an open standard used in many different application areas. Its grammar for a specific area can be described clearly and precisely and is extendable. There are already a lot of tools for parsing and manipulating XML data in many programming languages which means that a third-party module developer is not constrained in his choice of language.

We employ XML to describe the environment of the simulation, that is the Topology, Computing and Process layers. Especially for the topology we use the GraphML emerging standard [9].

We define a protocol for the communication of the DAP library with the engine as well as a protocol for the communication of the engine with the controllers and observers. By following these protocols it is easy to write a DAP library in a programming language other than `C++` since all the logic resides in the engine. Is is also simple to write another flavor of a graphical user interface; an example is a Java applet on a web page monitoring a simulation on the web server. We plan to provide libraries to ease the development of external modules.

We use also XML for recording all events that are processed. In this way the recorder data can be passed to an external tool for further processing, from pretty-printing (e.g., through stylesheets) to behaviour analysis (e.g., finding event patterns).

Finally, statistics can be written in either appropriate XML or in simple space-delimited text files (in gnuplot style).

7 Developing Distributed Algorithms with DAP

In order to run a simulation in DAP four types of executable programs are involved: an executable for the Engine, one for the Controller, one for the Observer(s), and one or more for the simulated processes. A user who wishes to simulate a process needs to create an executable for it. Note that two simulated processes may use the same executable. Even then, they may differentiate from each other since they may be run with different arguments/input and, also, they have different IDs.

A typical user program would access DAP through libDAP. Using libDAP is easy and the best way to demonstrate this is through an example. Consider the LeLann/Chang&Roberts algorithm [18] for leader election on a ring. Each process participating in the algorithm starts by sending its unique code to its next neighbor. Then, it forwards to the next neighbor every code it receives that is larger than any code it has seen so far. The process that receives back its own code becomes the leader, since its code went round the ring and is clearly the larger one. Finally, the leader sends a special message round the ring notifying the other processes that the election has ended. Implementing this algorithm in DAP is rather easy (see Fig. 4). Observe that to use DAP the program only needs to include a header file and have `main()` start with the declaration of a `DapSystem` variable. Through this variable it has access to all DAP services. Our aim was to make the use of these services as easy as possible, in order to keep the focus of the programmer not on using DAP but on developing the algorithm. Furthermore, a reader of the program should not be distracted from the algorithm by complex library calls nor should the length of the program increase in order to use DAP.

We can also easily support protocol stacking (i.e., to stack protocols in layers where the lower layers provide higher level functionalities for the higher layers). The user may implement protocol stacking in DAP by implementing the lower level protocols as different processes running on the same computing node with functionalities that are used by the higher level protocols.

```
#include "DapSystem.h"  /* required by DAP */

const int PREV = 0;   // port to previous node
const int NEXT = 1;   // port to next node
const string CODE = "codeTag"; // tag for message with code
const string END  = "endTag";  // tag for ending message with leader code

int main(int argc, char **argv)
{
    DapSystem dap(argc, argv);  // required by DAP, has to be 1st line

    /* initializations */
    StringMessage msg;  // use the string-based message implementation
    int myCode = atoi(dap.input("Enter unique code >0: ");
    int maxCode = myCode, inCode, senderID;
    string tag;
    int leader = -1;    // no leader initially
    dap.setColor(WHITE);

    /* send your code to the next process */
    msg << myCode;
    dap.sendToPort(NEXT, msg, CODE);
    while (leader < 0) { // run till you learn about the leader
        /* receive a code from previous process */
        msg.reset(); dap.receive(&senderID, msg, tag, true /* block */);
        if (tag == CODE) {
            msg >> inCode;
            if (inCode > maxCode) { // forward incoming code
                dap.sendToPort(NEXT, msg, CODE);
                maxCode = inCode;
            }
            else if (inCode == myCode) { // I am leader
                leader = myCode;
                dap.setColor(RED);
                /* notify others */
                msg.reset(); msg << leader;
                dap.sendToPort(NEXT, msg, END);
                do { // wait until the notification comes around
                     // (we need the loop for the non-FIFO channels)
                    msg.reset(); dap.receive(&senderID, msg, tag, true);
                }
                while (tag != END);
            }
        }
        else { // notification from elected leader (END)
            dap.setColor(YELLOW);
            msg >> leader;
            dap.sendToPort(NEXT, msg, END); // pass it on before exiting
        }
    }
}
```

Figure 4: The complete code for a leader election algorithm.

8 Concluding Remarks

We have presented the architecture and the design of the main components of the Distributed Algorithms Platform, and discussed several distinct features along with their implementation and functionality, as well as extensions upon which we are currently working. The main strengths of DAP are:

- The user programs in a standard programming language (`C++`) and the programs can be easily and efficiently ported to a real distributed environment.
- A user is able to control, monitor, and visualize the simulation of his algorithm through a Graphical User Interface.
- Many users can simultaneously monitor and visualize (through a lighter version of the GUI) the simulation of some algorithm from remote locations.
- The execution of the simulation can be distributed among several hosts (if required).
- Randomness can be introduced in all areas of the simulated distributed system.
- Scenarios can be defined for repetetive simulation of particular situations.

The current version of DAP may not be very efficient in simulating algorithms over very large networks (due the huge number of processes required). To address this issue, the current architecture should be enhanced. For example, if an algorithm invokes only non-blocking directives, it may be possible to run all the simulated process in one executable (since there is no danger that one process could block the whole simulation). We are currently investigating several alternatives to enhance the scalability of the system.

References

[1] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Pro. 4th ACM/IEEE International Conference on Mobile Computing and Networking* – MOBICOM'98, pp. 85-97, 1998.

[2] I. Chatzigiannakis, S. Nikoletseas, P. Spirakis. Analysis and Experimental Evaluation of an Innovative and Efficient Routing Approach for Ad-hoc Mobile Networks. In *Algorithmic Engineering* – WAE 2000, Lecture Notes in Computer Science, Vol. 1982 (Springer, 2000), pp. 99-110.

[3] I. Chatzigiannakis, S. Nikoletseas, P. Spirakis. An Efficient Communication Strategy for Ad-hoc Mobile Networks. In *Proc. 15th Symp. on Distributed Computing* – DISC 2001, Lecture Notes in Computer Science, Vol. 2180 (Springer, 2001), pp. 159-171.

[4] DSP. `http://helios.cti.gr/alcom-it/dsp`.

[5] O. Dahl, K. Nygaard. SIMULA — an Algol-based Simulation Language. *Communications of the ACM*, 9(9), pp. 671-678, 1966.

[6] The Esterel language. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.

[7] S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In *Foundations of Component-Based Systems* (G. T.Leavens and M. Sitaraman, eds), Cambridge University Press, 2000, pp. 285-312.

[8] E. Gamma, R .Helm, R. Johnson, J. Vlissides Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley, 1995

[9] The GraphML File Format. `http://graphml.graphdrawing.org/`

[10] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas, and R. Tan. Fundamental Control Algorithms in Mobile Networks. In *Proc. 11th ACM Symp. on Parallel Algorithms and Architectures* – SPAA'99, pp. 251–260, ACM Press 1999.

[11] G. Holland, N. Vaidya. Analysis of TCP Performance over Mobile Ad Hoc Networks. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing and Networking* – MOBICOM'99, pp. 219-230, ACM Press 1999.

[12] The IOA homepage. `http://theory.lcs.mit.edu/tds/ioa.html`.

[13] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark. Scenario-based Performance Analysis of Routing Protocols for Mobile Ad-hoc Networks In *Proc. 5th ACM/IEEE International Conference on Mobile Computing and Networking* – MOBICOM'99, pp. 195-206, ACM Press 1999.

[14] J. R. Jump. *NETSIM Reference Manual, Version 1.0.* Rice University, 1993.

[15] J. R. Jump. *YACSIM Reference Manual, Version 2.1.* Rice University, 1993.

[16] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* 3rd ed., McGraw-Hill, 2000.

[17] M. C. Little and D. McCue. Construction and Use of a Simulation Package in C++. In *C User's Journal*, 12(3), 1994. Also at `http://cxxsim.ncl.ac.uk/`.

[18] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[19] S. McCanne and S. Floyd. *ns Network Simulator.* `http://www.isi.edu/nsnam/ns/`.

[20] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[21] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing'93*, pp. 878-883, IEEE Computer Society, 1993.

[22] V. Sunderam, PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.

[23] P. Spirakis and C. Zaroliagis, "Distributed Algorithm Engineering", Chapter 10 in *Experimental Algorithmics - From Algorithm Design to Robust and Efficient Software* Eds. R. Fleischer, B. Moret, and E. Meineche-Schmidt, Springer-Verlag, 2002, pp. 197-228.

[24] B. Weiss, G. Gridling, U. Schmid, and K. Schossmaier. The SimUTC Fault-Tolerant Distributed Systems Simulation Toolkit. In *Proc. 7th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* – MASCOTS'99, 1999.

[25] The XML homepage at the W3 consortium. `http://http://www.w3.org/XML/`.