1. Distributed Algorithm Engineering

Paul G. Spirakis and Christos D. Zaroliagis

Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
and
Department of Computer Engineering & Informatics
University of Patras, 26500 Patras, Greece
Email: spirakis@cti.gr, zaro@ceid.upatras.gr

Summary.
When one engineers distributed algorithms, some special characteristics arise that are different from conventional (sequential or parallel) computing paradigms. These characteristics include: the need for either a scalable real network environment or a platform supporting a simulated distributed environment; the need to incorporate asynchrony, where arbitrary asynchrony is hard, if not impossible, to implement; and the generation of "difficult" input instances which is a particular challenge. In this work, we identify some of the methodological issues required to address the above characteristics in distributed algorithm engineering and illustrate certain approaches to tackle them via case studies. Our discussion begins by addressing the need of a simulation environment and how asynchrony is incorporated when experimenting with distributed algorithms. We then proceed by suggesting two methods for generating difficult input instances for distributed experiments, namely a game-theoretic one and another based on simulations of adversarial arguments or lower bound proofs. We give examples of the experimental analysis of a pursuit-evasion protocol and of a shared memory problem in order to demonstrate these ideas. We then address a particularly interesting case of conducting experiments with algorithms for mobile computing and tackle the important issue of motion of processes in this context. We discuss the two-tier principle as well as a concurrent random walks approach on an explicit representation of motions in ad-hoc mobile networks, which allow at least for average-case analysis and measurements and may give worst-case inputs in some cases. Finally, we discuss a useful interplay between theory and practice that arise in modeling attack propagation in networks.

1.1 Introduction

It is a common feeling among scientists, not only in the algorithms community, that a significant fraction of the research done in the algorithms area is eminently practical. However, only a small part of it is actually used. A suggested and also widely accepted remedy of this is that algorithmic research must include experiments and implementation if the field wants to have maximum impact.

In certain new fields much affected by current technology, the need of demonstration of practicality of algorithmic research is more intense. Such a field is that of *distributed systems*. These systems are ubiquitous today throughout business, academia, government, and home. Typically, they provide means to share resources and data. More ambitious distributed systems attempt to provide improved performance by attacking subproblems in parallel, and to provide improved availability in case of failures of some components.

The research in *distributed algorithms* tries to identify fundamental problems that are abstractions of those that arise in a variety of distributed systems, state them precisely, and then design and analyze efficient solutions. However, there are some important differences from the sequential case. First, there is not a single, universally accepted model of distributed computation – and there probably never will be – since distributed systems tend to vary much more than sequential computers do. Second, fundamental difficulties are introduced by three factors: *asynchrony, limited local knowledge*, and *failures.* The term asynchrony means that the absolute and even relative times at which events take place cannot always be known precisely. Also, since each computing entity can only be aware of information that it acquires, it has only a local view of the global situation. Computing entities can fail independently, leaving some components operational while others are not.

The explosive growth of distributed systems makes it imperative to understand how to overcome these difficulties. The fact that these difficulties are of a fundamental nature, led the theoreticians of distributed computing towards an effort to abstract and model their "negative" nature. In fact, the field of the theoretical aspects of distributed computing is full of lower bound and impossibility results. It is perhaps the field where the notion of an *adversary* to the solution is so well examined and modeled.

We believe that for all these reasons a systematic theory of *distributed algorithm engineering* should arise. By the term *distributed algorithm engineering* we mean the considerable effort required to convert theoretically efficient and correct distributed algorithms to efficient, robust, and easily used software implementations on a simulated or real distributed environment, usually accompanied by thorough experimentation, fine-tuning and testing. This conversion has to preserve the assumed properties and limitations of the distributed computing model. This in addition implies that the semantics of the implementation operations must agree with those assumed by the theoretical algorithm. Our experience indicates that such a conversion process may lead to improved distributed algorithms through perhaps the experimental discovery of behaviours or properties that were not exploited in the initial theoretical version of the algorithm.

The conducted experiments with implementations of distributed algorithms, henceforth *distributed experiments*, should evaluate algorithms by primarily providing them with "difficult", or "practical" inputs. It is exactly the rich collection of negative results about the "adversaries" of a distributed algorithm that allow the start of a systematic theory of the construction of "hard" instances, so useful for experiments. In addition, the need for detailed parameterization of the various complexity measures involved in any distributed problem, has led to a good understanding of various costs (such as number of messages, size and number of shared variables, number of faulty components, etc) which provides strong tools for the answer to the important question of "what to measure?" in a distributed experiment. The above idiosyncracies of distributed experiments justify well the creation of a whole new subfield of algorithm engineering. The current paper is a first contribution in this direction.

Our aim in this work is to address several methodological issues in distributed algorithm engineering. We do not attempt to cover every possible issue, but to address those which we find important. As explained above, we emphasize on issues that are not usually encountered when engineering sequential or parallel algorithms. Most of them are illustrated by case studies that are based on our own experience with developing simulators for distributed algorithms and experimenting with implementations of distributed algorithms on these simulators.

We start by discussing the need for a simulation environment which addresses the critical issue of scalability. Real distributed systems and algorithms are asynchronous. Incorporating asynchrony into a simulator is rather hard (if at all possible). To this end, we focus on a causality-affects relation which distributed experiments should obey and discuss advantages and disadvantages of known approaches to achieve it.

We then address the challenging issue of generating difficult (e.g., worst-case) inputs for implementations of distributed algorithms. We argue for two approaches, namely the construction of worst-case event schedules by mimicking impossibility arguments or lower bound proofs, and the use of game-theoretic notions (worst-case Nash equilibria) to construct test-sets which force the implemented algorithm to exhibit a nearly worst-case behaviour. To demonstrate these ideas, we present the experimental analysis of a pursuit-evasion protocol and an example of a lower bound proof for a shared memory problem which leads to worst-case schedules.

We then proceed to a particular interesting case of conducting experiments with algorithms for mobile computing. Two alternative models for mobile computing are discussed, the fixed backbone model and the ad-hoc model. The new element here is the question of how to implement motions. We discuss the two-tier principle that usually guides the fixed backbone model as well as a concurrent random walks approach on an explicit representation of motions in ad-hoc mobile networks which allow at least for average-case analysis and measurements, and may give worst-case inputs in some cases.

Finally, we consider a rather general model for modeling attacks in computer networks and discuss the development of protocols for propagation of attacks under this model. The development of such protocols turns out to be an interesting case of distributed algorithm engineering as both analytic and experimental methods are used which are tied to each other.

1.2 The Need of a Simulation Environment

In this section, we will argue about the need for a simulation environment which addresses the crucial issue of scalability. We first attempt to formally define the simulator and then give an overview of existing systems.

Distributed applications code runs usually on rather huge networks of possibly heterogeneous local machines. Even if one manages to control such an environment for conducting experiments, still several important questions cannot be answered due to the implied restrictions of available technology. Perhaps the most important one is that of *scalability*: given that a distributed algorithm behaves "well", for example, on a network of ten machines, how will it behave on a much larger network? Thus, this critical issue of scalability of distributed solutions can be experimentally treated only via *simulations*. It is a nice byproduct of distributed algorithmic practice the fact that simulations and simulation environments are themselves extensively studied and formalized. The crucial idea here is that the simulator executions should not alter the nature of executions on the "real" (or envisioned) system. We can capture this via some definitions by adopting the framework in [1.3].

We view a *distributed system* as a collection of a set of *nodes* or *processors*, a communication system $\mathcal{C}$ linking the nodes, and the external *environment* $\mathcal{E}$. Usually the environment $\mathcal{E}$ and the communication system $\mathcal{C}$ are not explicitly modeled but are given as problem specifications, which impose conditions on their behaviour. A *node* or *processor* is a (rather) hardware notion. On each node there are one or more (software) *processes* running. Let us, for the sake of definition of simulations, restrict our attention to the situation where processes are organized into a single *stack of layers* and that there are the same number of layers on each node. Each layer communicates with the layer above it and the layer below it. The bottom layer communicates with $\mathcal{C}$ and the top layer communicates with $\mathcal{E}$.

Each *process* is modeled as an automaton with a (possibly infinite) set of states. Transitions between states are triggered by the occurrence of *events* of the process. *Events* are inputs or outputs that come from (or go to) the layer above or below. A *configuration* of a distributed system specifies a state for every process on every node. An *initial configuration* contains all initial states. An *execution* of a distributed program is a sequence $C_0 \Phi_1 C_1 \Phi_2 C_2 \cdots$ of alternating configurations C_i and events Φ_i beginning with a configuration and, if finite, ending with a configuration. An execution must satisfy four conditions:

1. C_0 is an initial configuration.
2. For every $i \geq 1$, event Φ_i is enabled in configuration C_{i-1} and configuration C_i is the result of Φ_i acting on C_{i-1}. In more detail, every state is the same in C_i and C_{i-1} except for the (at most two) processes for which Φ_i is an event. The states of these processes change according to the transition functions of those processes.

3. For every $i \geq 1$, if Φ_i is not a node input, then $i > 1$ and Φ_i is on the same node as event Φ_{i-1}.
4. A node input does not happen until all other events have acted and no more are enabled.

The last two conditions are stated just to guarantee atomicity with respect to events on different nodes. A node is triggered into action by the occurrence of an input either from the external environment, or from the communication system. The trigger causes a "chain reaction" of events at the same node, and this occurs atomically, until no more events are enabled, other than node inputs. In fact, there are many ways to state (or even omit) conditions (3) and (4) if the implementation guarantees atomicity of events on different nodes via some other mechanism, for example, via the event generation scheme or via the scheme that assigns durations to steps of processes and delays to messages according to a global simulator (virtual) clock.

The *schedule* of an execution is the sequence of events in the execution. Given execution a, let us denote by $top(a)$ (resp. $bot(a)$) the restriction of the schedule for a to the events on the interface of the top (resp. bottom) layer. An execution a is then said to be *correct for communication system* $\mathcal{C}$ if $bot(a)$ is an element of the allowable sequences of inputs/outputs of $\mathcal{C}$. An execution is *fair* if every event (other than a node input) that is continuously enabled, eventually occurs. This ensures that executions do not halt prematurely, while there is still a step to be taken. An execution a is *user-compliant for problem specification* P if, informally speaking, the environment satisfies the input constraints of P (if any). The details of the input constraints will naturally vary depending on the particular problem. An execution is called $(P, \mathcal{C})$-*admissible* if it is fair, user-compliant for the problem specification P, and correct for the communication system $\mathcal{C}$.

We are now ready to define what a simulator should do. Let us denote by communication system C_1 whatever is available for our experiments. We wish this to simulate some (larger or envisioned) communication system C_2. For such a simulation we then demand the existence of a collection of processes called *Sim* (the simulation program) which must satisfy three laws:

1. The top interface of *Sim* is the interface of C_2.
2. The bottom interface of *Sim* is the interface of C_1.
3. For every (C_2, C_1)-admissible execution a of *Sim*, there exists a sequence σ of events in the set of sequences of events of C_2 such that $\sigma = top(a)$.

Informally, we run our simulation on top of the (available for experimentation) communication system C_1 and this produces the same appearance to the external environment as does the (envisioned) communication system C_2. This is illustrated in Figure 1.1.

Let us define the *richness* of (Sim, C_1) as the set of all possible C_2 communication systems that can be simulated by it. It is then apparent that our experimental environment (Sim, C_1) for distributed experiments should be

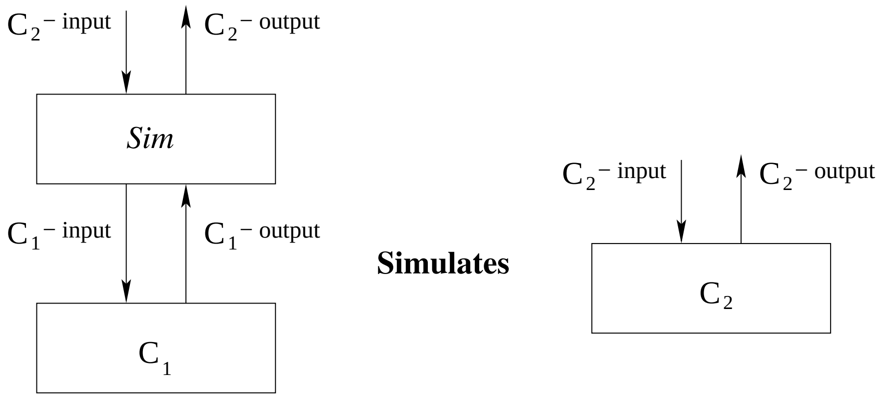

Fig. 1.1. The simulation of a communication system C_2.

rich enough to include C_2 communication systems which are, for example, scalable C_1 communication systems or extensions or systems with stronger technological capabilities. There are several such environments, some of which we discuss next.

1.2.1 An Overview of Existing Simulation Environments

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 three such environments: DSP [1.11], IOA [1.20], and DAP [1.10]. On the other hand, there are several environments for the simulation of 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 [1.32], YACSIM [1.22], and SimUTC [1.42]. 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, DAP) 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. In the rest of this section, we shall give a brief overview of the above mentioned simulation environments.

Ns (network simulator) [1.32] 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 animation tool (nam) for animating the simulation accompanies ns.

The IOA project [1.20] provides a formal language for describing processes that are modeled using I/O automata and a toolset (currently under development) which will provide support for the development, analysis, and simulation of IOA programs. The toolset is developed in Java. To model a distributed algorithm or system, a user has to program in the IOA language. (A similar effort was done earlier with the Esterel language [1.12].)

The Distributed Systems Platform (DSP) [1.11] is a software platform that has been designed for the implementation, simulation, and testing of distributed algorithms. It offers a set of subtools which allow the researcher and the algorithm designer to work under a familiar graphical and algorithmic environment. DSP provides a set of simple, algorithmic languages (DSPL) which can describe the topology and the behaviour 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. The DSP tool has been implemented in C.

YACSIM [1.22] is a discrete event simulator implemented in C that provides a basic set of C subroutines (which model processes, events, queues) that the user can link with his/her 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 is normally used as a base to produce more sophisticated simulators. For example, it was used to produce NETSIM [1.21], a general purpose interconnection network simulator for parallel architectures.

SimUTC [1.42] is a simulation toolkit intended to develop simulations of algorithms for the specific problem of clock synchronization. It is round-based and built on C++SIM [1.29], a toolkit written in C++ that implements facilities provided in SIMULA [1.9]. It uses threads and is a process-oriented, continuous-time discrete-event simulator. It is interesting that the C++SIM developers observe that C++ compilers produce much more efficient code than SIMULA, thus resulting in faster simulations.

The Distributed Algorithms Platform (DAP) [1.10] is a software platform, currently under development, aiming to support the implementation, simulation, and testing of distributed algorithms. It is implemented in C++ using LEDA [1.33]. To transfer the full power of LEDA to distributed experiments and implementation of distributed algorithms, DAP will be provided as a LEDA Extension Package. The major goal of DAP is to provide a homogeneous environment for the simulation of distributed algorithms, regardless of whether they are designed for wired or for wireless networks. It allows the algorithm designer to implement a protocol using a standard programming language (C++), along with the primitives of the DAP library, and hence waive the need for re-writing an existing application program as well as the need for an interpreter. DAP also provides a graphical user interface that allows the execution of simulations, protocol animation, as well as correctness checking.

1.3 Asynchrony in Distributed Experiments

Notions of causality and time play an important role in the design of distributed algorithms. It is often helpful to know the relative order in which events take place in the system. This knowledge can be achieved (and experimentally dealt with), even in *totally asynchronous systems* that have no way of measuring the passage of real time, by observing the causality relations between events.

In many systems, processors have access to real-time measuring devices, for example, to hardware clocks, or by tuning in to a satellite clock, or by reading the time across a communication network. In such cases the experiments become easier, since we only have to provide the logical equivalent of a "global time".

Since executions of a distributed system are sequences of events, they induce a *total order* on all possible events. However, this way of describing executions is experimentally painful since it requires the (frequently intolerable) overhead of submitting our experiment to all possible sequences of events! This is wasteful since it is possible that two computation events by different nodes, which may not influence each other, to be nonetheless arbitrarily ordered by the execution. What is important for our experiment to capture is the *structure of causality* between events.

Consider two events by different processors (nodes) – possibly simulated in our experiment. The only way for one processor to influence another processor is by *sending information* (a *message*) to the other processor. But also note that events can causally influence each other indirectly through other events. Hence, it seems that a successful distributed experiment should capture this essential causality relationship for *every* execution. Consequently, a *causality-affects* relation for execution a is defined as follows (see [1.3, Ch. 8] and also [1.28]).

Given two events Φ_1 and Φ_2 in a, we say that Φ_1 *causally-affects* Φ_2 *in* a, denoted by $\Phi_1 \stackrel{a}{\Longrightarrow} \Phi_2$, if one of the following holds:

(i) Φ_1, Φ_2 are events in the same (sequential) process p_i and Φ_2 follows (occurs after) Φ_1 in a's part of p_i.
(ii) Φ_1 is the "send" event of passing information I from process p_i to another process p_j and Φ_2 is the "receive" event of information I by p_j.
(iii) There exists an event Φ such that $\Phi_1 \stackrel{a}{\Longrightarrow} \Phi$ and $\Phi \stackrel{a}{\Longrightarrow} \Phi_2$.

We conclude that distributed experiments should *respect* the causality-affects relationship.

Even if processors cannot "observe" the above relationship, the experiment has to observe it. Till now, there are two approaches (both unsatisfactory for reasons that we will explain) for dealing experimentally with this. The first approach is to supply the experiment with an *event generator* which produces (sequentially in time) events for various processes that respect causality. The second approach is to assign "time durations" (or delays) to each local process step and to each send-receive event in the experiment. These delays can be just integers and may not necessarily relate to the local simulation hardware clock, but they must respect the specifications of the (possibly hypothetical) system on which the experiment runs via the simulator.

The trouble with both approaches is that usually the specifications of the hypothetical system, on which the distributed algorithm runs, allow for many (sometimes a vast number of) total orders of causally related events. An example is the specification of a totally asynchronous system, where no relation is given in advance between durations of local processor steps or message delays.

The usual experimental answer here is the use of random number generators in order to assign local or message event durations. However, even this approach suffers in two directions. First, each random number selection must specify an interval of possible numbers. This, theoretically, restricts the number of possible event orderings. Second, processes might try to draw conclusions about asynchrony by monitoring locally their event durations. Such a (statistical) monitoring should not draw conclusions about biases or about the mechanism of (pseudo) randomness. Cryptographically secure generators might be an answer to this problem [1.5].

In any case, the experimental implementation of the degree of asynchrony, specified by the hypothetical system specifications, should at least produce those admissible sequences of events that can demonstrate the worst-case behaviour of the implemented algorithm when all other inputs are fixed. This issue is further investigated in the next section.

1.4 Difficult Input Instances for Distributed Experiments

In this section we address the problem of how to generate "hard" input instances for experiments on distributed algorithms or protocols. We focus on two approaches:

(a) An *adversarial-based approach*, taking as point of departure the rich literature about adversarial arguments in distributed computing. These arguments often lead to proofs of impossibility results for the computation of certain tasks or to lower bounds on the (worst-case) performance of any distributed algorithm for a certain problem. We note that quite frequently such arguments are indeed *adversarial constructions* in the sense that they propose particular execution orders of certain events (among the admissible schedules) and/or particular fault patterns that can be produced effectively in an experiment and that have the property of driving any distributed algorithm to its limits as far as worst-case performance is concerned. In fact, many impossibility scenarios can be modified suitably to create hard inputs for the experiments.

(b) A *game-theoretic approach*; namely, to view the distributed protocol as a game in cases where processes or agents may act selfishly or compete to each other for resources. The goal is to select (if computationally possible), among the possibly many Nash equilibria for such games (which are a well-accepted characterization of "rational" behaviour in competition situations), those equilibria that are as worst as possible according to a global system, or *social cost*, criterion. Then, the computed worst-case equilibria strategies are used as the "hard" input instances in the experiments.

In the rest of this section we shall elaborate on these two approaches.

1.4.1 The Adversarial-based Approach

Any distributed algorithm has to overcome a variety of *adversarial* system behaviours. For example, processes may fail (and perhaps later recover), their states can become corrupted, or even they can behave maliciously. Communication channels can fail, lose or delay messages, or deliver them out of order. Also, shared (memory) objects may fail to respond.

Such behaviours are captured by the notion of an *adversary* to the algorithm. The adversary can select the failure patterns and/or the schedules of events among the admissible schedules. The adversary may control a subset of system's processes. Such processes might relay false information (even deliberately) and can be allowed to conspire.

The precise characterization of the power of the adversary is crucial, because its consequences are either *impossibility results* (that is, no distributed algorithm can achieve certain goals under such adversaries), or *lower bounds* in worst or average case performance which cannot be improved by any distributed algorithm.

Among the earliest impossibility proofs in distributed computing is the impossibility result for the achievement of distributed consensus in an asynchronous system of processes even with only one faulty process, that is, the well-known result by Fischer et al. [1.13]. (Perhaps the oldest impossibility result for agreement of processes on a value was given by Pease et al. [1.37].)

In the paper by Fischer et al. [1.13], the authors were the first to use a *valency argument* to show that consensus achievement is impossible in a totally asynchronous message-passing system which is allowed to tolerate just one process fault. This fault can be the simplest one, i.e., the faulty process fails by halting permanently at some point (fail-stop model). Valency arguments have become the most widely-used techniques for impossibility proofs in distributed computing. We now give an outline of the valency argument.

Recall that a *configuration* is basically a "snapshot" of a distributed system during the execution of an algorithm. It consists of the state of every process and of the surrounding environment (e.g., messages in transit). A configuration of any consensus algorithm is called *univalent* if every possible execution continuing from that configuration gives the same output value, and *multivalent* otherwise. In the case where the possible output values are just two, then the configuration is called *bivalent*. For example, in the *binary consensus* problem all input values to the processes come from $\{0, 1\}$. To achieve consensus, there are two correctness properties that must be satisfied.

1. Agreement: the output values of all processes are identical.
2. Validity: the output value of each process is the input value of some process.

Now, notice that in the case of input values from $\{0, 1\}$, any consensus protocol must have a bivalent initial configuration. Let e be any event applicable to a bivalent configuration C, let D be the set of configurations reachable from C without applying e, and let $D^* = \{C'' :$ configuration C'' follows from configuration C' by applying e and $C' \in D\}$.

Then, it is proved in [1.13] that D^* contains a bivalent configuration (the proof of this, rather intuitive, statement is very technical, but beautiful). Consequently, any deciding (on a value) execution must go from a bivalent initial configuration to a univalent one, which in turn implies that there should be some single step that goes from a bivalent to a univalent configuration. In [1.13] a particular way is proposed for an execution that avoids such steps and thus leading to an execution that never decides. The execution is constructed in *stages*, starting from an initial configuration. Each stage has one or more steps of some processes. A queue of processes is maintained (initially in an arbitrary order). For each process, a queue of incoming messages is also maintained. The stage ends with the first process in the queue of processes executing a step, in which, if its message queue was not empty at the start of the stage, then its earliest message is received. This process is then moved to the back of the queue of processes.

Note that this execution can be easily implemented in an experiment, and that in an infinite sequence of stages, every process takes an infinite number of steps and receives every message sent to it. In [1.13] it is then shown that this execution avoids a decision ever being reached, because it always produces bivalent configurations.

Although valency arguments are the most well-known techniques to show impossibility results, other arguments (for example, based on algebraic topology [1.18]) have also been used.

We note that in the valency arguments it is crucial for the adversarial scheduler to select when to schedule a particular process (in order to destroy consensus). Most lower bound arguments also make use of such adversarial schedules of events. Based on this important remark, we propose that such adversarial schedules should be tried (if possible) in a distributed experiment. Then, the experiment will reveal the worst-case behaviour of the proposed protocol under test. Of course, adversarial schedules are not always easy or possible to construct, but the impossibility or lower bound proofs in most cases give strong hints. We illustrate the method through an example.

1.4.1.1 An Example of an Adversarial Schedule. In this section, we shall present an example of an adversarial schedule which is easy to implement.

Suppose that we want to experiment with an algorithm A that solves the following problem, called the *write-all problem* [1.23, 1.25]: P processes are given, all having access to a shared memory (that is, to an array $M[1..\infty]$). Let the first N shared memory locations be called the *write-all array.* All processes are assumed to work in complete synchrony, that is, in each global time unit each process takes a step. The adversary can cause arbitrary process failures and restarts. The problem is to provide a distributed algorithm which, at termination, has managed to *touch* (mark) each position of the write-all array by some process (initially all memory is untouched). The performance measure here is the total number of steps of all processes until this is done. This is called the *work* of the algorithm.

To test such an algorithm A, we suggest the following adversarial failure/restart schedule as proposed in [1.6]. Consider each global step. Let $U > 1$ be the number of untouched array elements (i.e., the elements that no process succeeds in writing to them). For as long as $U > P$ the adversary induces no failures. The work needed to touch $N - P$ array elements when there are no failures is at least $N - P$. As soon as a process is about to touch the element $N - P + 1$, making $U \leq P$, the adversary fails it and then restarts all P processes. For the upcoming cycle, the adversary examines the algorithm's implementation to determine how the processes are assigned to touch array elements. The adversary then lists the first $\lfloor U/2 \rfloor$ untouched elements with the least number of processes assigned to them. The total number of processes assigned to these elements cannot exceed $\lceil P/2 \rceil$. Subsequently, the adversary fails these processes and allows all others to proceed. Therefore, at least $\lfloor P/2 \rfloor$ will complete their step having touched no more than half of the remaining untouched array locations. This strategy of failures/restarts can be continued for at least $\log P$ global steps. Then, the work that A performs is at least $N - P + \lfloor P/2 \rfloor \log P = N + \Omega(P \log P)$.

Note that the above schedule of failures/restarts can be easily constructed in the experiment, given any algorithm A and its implementation. Note also that the lower bound to the required work does not count how much work is needed for the processes to read and locally process (without touching the write-all array) the entire shared memory. Thus, such a schedule will cause the implementation of any algorithm A to perform work bounded from below by $N-P+\lfloor P/2 \rfloor \log P$. Most algorithms A will actually do more work, since usually processes can read only a constant number of shared memory locations at each step. Let $L = N - P + \lfloor P/2 \rfloor \log P$ and let $W(A)$ denote the actual work performed by A's implementation. We can then use $W(A) - L$ as a measure of how work-efficient A is for such work-demanding schedules.

Logarithmic lower bounds on the time of any synchronous execution for deterministic write-all were first derived in [1.24].

1.4.2 The Game-Theoretic Approach

Distributed systems often invoke a set of independent *selfish* and *antagonistic agent* processes trying to share a common resource. This situation evokes game theory and its main concept of rational behaviour, the *Nash equilibrium*: in an environment in which each agent is aware of the situation facing all other agents, a Nash equilibrium is a combination of choices (deterministic or randomized), one for each agent, from which no agent has an incentive to unilaterally move away. The ratio between the worst possible Nash equilibrium and the global optimum, called *coordination ratio*, was first defined in [1.27]. Some upper bounds for this ratio and the structure of worst-case Nash equilibria for a very simple routing problem were given in [1.31].

An alternative way to derive "difficult" behaviours (schedules of events) for distributed experiments is to use such game-theoretic ideas. We can sometimes consider the competition between a distributed algorithm and an adversary as a game of possibly many rounds of moves of the opponents. Worst-case Nash equilibria (with respect to some optimization criteria) may then be examined and we suggest them as interesting behaviours to be tested experimentally.

We motivate the above approach by a very simple problem of pursuit-evasion. Several agents moving along neighbor vertices of a graph (network) G are looking for a fugitive. The fugitive is eliminated when it coincides with an agent at a vertex. The agents cannot "see" further away from their current location. The way the graph (network) operates is a sequence of *rounds*, each of the form $R = (T, F, S)$, where T is an *agent's target phase*, followed by a *fugitive motion phase* F, followed by an *agent's motion phase* S.

Any protocol (strategy) for the agents implements T and S as follows. Let $v(a)$ be the current position, i.e., vertex in G, of agent a, and let $N(v(a))$ denote the set of neighbors of $v(a)$ in G. Let also $t(a)$ be a variable which can be written by the agent and can be read by the fugitive under some conditions that we explain next. We call $t(a)$ the *target variable* of agent a. T is implemented by setting $t(a)$ to be a vertex in $\{v(a)\} \cup N(v(a))$. S is implemented by setting the next position $v'(a)$ of the agent to be the value of $t(a)$.

Any strategy for the fugitive, f, implements F as follows. Let $v(f)$ be the position (vertex in G) of f just before round R, and let $N(v(f))$ denote the set of neighbors of $v(f)$ in G. Each edge (x, y) in G (in direction from x to y) is equipped with a queue, called the *input channel of vertex* y, where an entity located at x can put information. This information can subsequently be read by y. The fugitive can read (and store in its local memory) the value of one input channel $c(u)$ for each $u \in N(v(f))$. The value of each $c(u)$ is defined as follows: if there is an agent a in R with $v(a) = u$ and $t(a) = v(f)$, then $c(u) = v(f)$; otherwise, $c(u)$ is undefined. Thus, the fugitive is allowed to be "warned" about the next position of any agent only when that position is the next position of the fugitive f (we say that the fugitive has a limited sense of approaching agents). Let $\mathcal{C}_i = \{c(u) : \forall u \in N(v(f)) \text{ where } v(f) \text{ is the position of } f \text{ in round } i\}$ be the set of all "warnings" that f got in round i. Then, the ordered tuple $H_R(f) = \langle \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_R \rangle$ of channel values of all rounds till R denotes the *history* of all "warnings" that f got up to round R. Consequently, the fugitive's strategy F decides on a next position for f (which must be a neighbor of $v(f)$ in G) based on $H_R(f)$.

Simple randomized protocols for catching the fugitive were presented in [1.39, 1.40]. In their general structure, these protocols suggest that agents are partitioned into two sets: the *traps*, which stay immobile (hidden therefore) at some random vertices of the graph, and the *searchers*, i.e., agents continuously performing independent random walks. Note that the above stated model for the strategies of the agents and the fugitive does not allow the fugitive to "sense" neighboring traps, since their intention variable does not change and thus no information comes via the related channel. Actually, f cannot distinguish between traps and vertices which are neighbors and agent-free.

In this game, good strategies for the fugitive should allow it to stay alive and not be eliminated for as long as possible. Fugitive motion around some chosen cycle in the graph is (together with the way agents act) a Nash equilibrium. The fugitive, while not caught, has no benefit in not following the cycle. The game ends almost surely only if the traps can re-randomize their locations from time to time [1.40]. Since long cycles of fugitive's movement have higher probability to encounter a trap, we conclude that short cycles of such a movement are worst-case equilibria in the sense that they extend the game's duration. Note that strategies which force the fugitive to stay at some vertex forever (after some initial motion) are not good, since random walks will hit any of those positions in short expected time.

A preliminary set of experiments with such strategies (that is, short cycles in the graph for motions of the fugitive) was performed in [1.15], while the game was theoretically analyzed in [1.39, 1.40]. The experiments indeed demonstrated the longest durations of the game for such fugitive strategies. These experiments were performed on the DSP tool [1.11] and are as follows. First note that if the fugitive is memoryless (or has only a fixed-size memory), then its best strategy is to randomly-avoid approaching agents, that is, when it senses an approaching agent, it should choose randomly any way to go except for the edge via which the agent is approaching. Hence, the experiments become much more interesting when the fugitive can store and remember non-fixed parts of the graph. The fugitive initially wanders in the graph until it discovers a small-sized cycle. Then it stops wandering and starts moving along the cycle.

Various different graph topologies, including regular, irregular and random graphs, were considered in [1.15], each consisting of 100 vertices and the same number of expected edges (about 1000). The protocol was initialized with 5 traps and 2 searching agents. The traps were re-randomizing their positions periodically. The performance was measured in extermination time, that is, after how many simulation rounds (excluding the rounds required for the traps to take their positions) the fugitive falls into a trap. The reported experiments indicated that the cycling fugitive lasts much longer (about 40 times) than the randomly-avoiding fugitive. Experiments were also conducted with various numbers of traps which indicated that when the number of traps is doubled, the extermination time drops almost linearly. A last set of experiments was performed with varying time periods, called *epochs*, for re-randomization of the location of the traps. The experiments showed that the extermination time grows almost linearly with the increase in the epoch size. In all cases the experiments considerably helped the fine-tuning of the parameters of the experimentally best strategy of the agents (number of traps, duration of epochs) against the worst-case equilibrium strategy of the fugitive.

We note that the approach presented above has already led other researchers to design fugitive strategies and agent strategies in similar ways, by especially following the game-theoretic paradigm in slightly different games where the fugitive is completely blind [1.2].

1.5 Mobile Computing

The major technological advances in mobile networks have recently motivated the introduction of a completely new computing environment called *mobile* (or *nomadic*) *computing* [1.19, 1.38].

Mobile computing is a special type of distributed computing that is characterized by four kinds of constraints which make the design of mobile information systems a highly complex task:

(i) Mobile elements have poor resources compared to static elements.
(ii) Mobile elements rely on a finite energy source (battery).
(iii Mobility is inherently hazardous regarding damage or physical security loss.
(iv) Mobile connectivity is highly variable in performance and reliability.

Hence, information access as well as fundamental distributed computing problems (e.g., leader election) have to be re-considered in the new setting. As a consequence, new approaches are usually required for the effective solution of these problems in order to develop a dependable and efficient mobile information system.

Until now, two basic models have been proposed for mobile computing: the *fixed backbone* model and the *ad-hoc* model. The *fixed backbone* model assumes that a fixed infrastructure of support stations with centralized network management is provided in order to ensure efficient communication. Communication is done through the support stations which serve a certain geographical area in which mobile hosts are moving. The fixed backbone model is motivated by the current status of pragmatic mobile networks.

The *ad-hoc* model assumes that mobile hosts can form temporary networks, called *ad-hoc networks*, without the aid of any fixed infrastructure or centralized administration. Communication between two hosts can be achieved through other mobile hosts which participate in the ad-hoc network and are willing to transfer packets for them. The ad-hoc model is motivated by the need for rapid deployment of mobile hosts in an unknown terrain (e.g., emergency services in a disaster area), where there is no underlying fixed network infrastructure either because it is impossible or very expensive to create it, or because it has become unavailable.

The above imply that the design, implementation, testing and verification of distributed protocols for mobile computing requires extension of software (simulation) platforms that are designed to support development of distributed algorithms on (classical) fixed networks. A simulation environment for mobile computing should be able to capture the notions of a *mobile process* (can be viewed as a virus or an agent), the *motion* of a process or host (which allows it to either migrate from cell to cell or to follow any course in a given space), the *energy* of a process or host, and the *channels* (which except for their bandwidth and latency could be further characterized by their frequency spectrum as well as by communication interference). All these are crucial in the design of a simulation platform for mobile computing (for example, they are taken into account in the Distributed Algorithms Platform [1.10]).

Perhaps the most important of the above notions is that of *motion* of processes or hosts. Two ways of capturing motions have appeared in the literature: the explicit and the implicit representation. The *explicit* representation provides a definition of the space of possible motions and also a definition of a trajectory of each host in that space. The *implicit* representation provides a dynamic graph of possible direct communications among hosts, and the way this graph changes with time. The change of communication edges is assumed to occur due to the motion of hosts.

One of the most interesting cases is to deal with systems in which fast motions of processes or hosts are allowed. In such cases, the implicit representation has major inherent analysis problems and limitations, since in such dynamic graph models protocols usually try to maintain network structures (e.g., connectivity, multiple paths, etc), but the time to allow information to propagate for the modification of these structures is not always comparable with the speed of change of the network.

Consequently, in the rest of this section, we will focus on the explicit representation of motions and will address certain methodological issues that arise in distributed algorithm engineering regarding implementation and experimentation of algorithms for mobile computing. We shall address these issues through two case studies, one for the fixed backbone model and another for the ad-hoc model. The former concerns the problem of counting the number of mobile hosts in a mobile network, while the latter concerns the fundamental problem of establishing point-to-point communication between two mobile hosts. Before diving into the case studies, we shall discuss the two models in more detail.

1.5.1 Models of Mobile Computing

The *fixed backbone* model assumes two distinct sets of entities in a mobile network: a large number of *mobile hosts*, and a relatively small number of more powerful, fixed hosts. The fixed hosts and the communication paths between them constitute the *static* or *fixed* (part of the) network. The geographical area that is served by the fixed network is divided into smaller regions called *cells*. Each cell is served by a fixed host, also referred to as the *mobile service station* (MSS) of the cell. An MSS communicates with the mobile hosts within its cell via a wireless medium of low bandwidth. Host mobility is represented in this model as migration of mobile hosts between cells; each mobile host belongs to only one cell at any time instance.

We now discuss how mobile hosts exchange messages and which is the incurred cost in the fixed backbone model. There are two types of messages: point-to-point messages between any two MSSs, and messages between a mobile host and its local MSS. Let the cost of a former message be C_f and the cost of a latter message be C_w. Assume that a mobile host h_1 wants to send a message to another mobile host h_2. The host h_1 sends first the message to its local MSS, which forwards this message to the local MSS currently serving h_2. Since, however, the location of a mobile host is neither fixed nor universally known to the network, the local MSS of h_1 needs first to determine the MSS which currently serves h_2 and then transmit the message. This incurs an extra *search cost* C_s for each message transmission. A reasonable assumption to consider [1.4] is that $C_s = aC_f$, where a is a constant depending on the location management strategy used. Suppose that m mobile hosts are moving throughout a fixed base station mobile network $G = (V, E)$ consisting of $n = |V|$ nodes (corresponding to the MSSs) and $|E|$ edges (representing the point-to-point direct communications between the MSSs). Note that $|E| = O(n^2)$ in the worst-case and that usually $m \gg n$. Let D be the diameter of G.

Then, $C_s = O(D)$ and a message between two mobile hosts incurs a cost of $2C_w + C_s$.

The *ad-hoc* model assumes that mobile hosts can form temporary networks, called *ad-hoc networks*. An *ad-hoc mobile network* [1.19] is a collection of mobile hosts with wireless network interfaces forming a temporary network without the aid of any established infrastructure or centralized administration. In an ad-hoc network two hosts that want to communicate may not be within wireless transmission range of each other, but could communicate if other hosts between them are also participating in the ad-hoc network and are willing to forward packets for them.

1.5.2 Basic Protocols in the fixed backbone model

A fundamental problem in any network is to count the number of available processes or nodes. In the case of mobile networks, this *counting problem* retains its importance: the knowledge of how many mobile users are currently connected is generally valuable and can be used both by the control and the application level of the network.

Two algorithms for the counting problem in the fixed backbone model were implemented and experimentally compared in [1.16]. The first algorithm in that paper is a simple modification of an existing algorithm for the counting problem in a fixed network [1.41, Ch. 6]. The second is a more efficient algorithm presented in [1.17].

In the rest of this section, we shall discuss the implementation and experimental evaluation of these algorithms on the Distributed Systems Platform [1.11]. In particular, we discuss the main issue of modeling the speed and the type of movement of the mobile hosts.

For the counting problem, it is assumed that one of the mobile hosts (the *initiator*) wants to find the number of the mobile hosts (m) in the network. It is further assumed that: the communication between the MSSs is based on the asynchronous timing model; an operational mobile host responds immediately to messages broadcasted by its local MSS; each mobile host has its own distinct identity; the set of mobile hosts does not change during the execution of a counting algorithm.

The first algorithm considered in [1.16], called the *virtual topology algorithm* (VTA), is based on the distributed execution by the mobile hosts of a counting algorithm for fixed networks. The VTA algorithm is based on the assumption that the mobile hosts are willing to control by themselves the execution of the algorithm by avoiding the participation of the MSSs (this may be necessary if some protocol requires computational power that increases the overhead of MSSs). The counting algorithm for fixed networks used is the *Echo protocol* given in [1.41, p.190]. It is based on the centralized wave paradigm [1.41, Ch. 6]. There is one initiator process and all other are non-initiators. The initiator floods token messages to all processes and eventually receives confirmation from all processes. The initiator sends messages to all its neighbors. Upon receipt of the first message, a non-initiator forwards messages to all of its neighbors, except the one from which the message was received and which marks as its parent and the corresponding link as its parent link. It is easy to see that parent links define a spanning tree of the fixed network. When a non-initiator has received messages from all its neighbors, it sends an "echo" message to its parent. When the initiator has received a message (either an echo or a flooding message) from all its neighbors, it terminates. The application of the echo protocol in a mobile network implies that some kind of virtual topology is defined on the mobile hosts. By assuming that the virtual topology has $O(m)$ edges, the total cost of the VTA algorithm is $O(mC_w + DmC_f)$, where D comes from the search cost in the fixed network.

The VTA algorithm has two drawbacks: (i) the high cost of message transmissions in the fixed network; and (ii) it requires the participation of every mobile host in the virtual topology. The latter is crucial, since no mobile host is allowed to disconnect during execution of VTA and which in turn brings into play another parameter: the total execution time of the algorithm which should not be large since otherwise it would increase the consumption of battery power in the mobile hosts.

The second algorithm implemented in [1.16] is a new counting algorithm, especially designed for the fixed backbone model, and presented in [1.17]. The algorithm is based on a common guiding principle of distributed protocols in the fixed backbone model, called the *two tier principle* [1.1], and consequently named the *two tier algorithm* (TTA). The idea of this principle is that the computation and communication costs of an algorithm should be based, as much as possible, on the fixed portion of the mobile network.

The TTA algorithm is also based on the execution of the Echo protocol. The execution is started by the *initiator mobile host* which broadcasts a "count" message (afterwards, this initiator does not respond to any "count" message). The (non-initiator) mobile hosts receive "count" messages from their local MSS and respond with "count-me" messages in order to be counted by the MSS. The rest of the TTA algorithm is executed by the fixed part of the network (i.e., by the MSSs). Let the *initiator MSS* be the MSS serving the initiator mobile host. The part of the algorithm executed by the MSSs is as follows.

1. The initiator MSS broadcasts a "count" message in its cell and then spreads along the fixed part of the network the request for counting using the Echo protocol. The protocol starts by sending "count-tok" messages to all adjacent MSSs.
2. An MSS, upon receiving a "count-tok" message, broadcasts a "count" message to its cell and waits to collect answers from mobile hosts in this cell. The MSS also forwards a "count-tok" message to its neighbors to continue the execution of the Echo protocol. When the MSS receives

a "count-me" message, it increases a local variable sz which stores the number of counted mobile hosts in its cell. If the MSS receives a "join" message (from a mobile host joining its cell), it broadcasts again a "count" message. After the completion of the Echo protocol, all MSSs have been informed about the execution of a counting algorithm and have broadcasted a "count" message in their cell.
3. The initiator MSS starts a second execution of the Echo protocol by sending a "(size-tok,0)" message to its neighbors aiming at collecting the sz variables from all MSSs to the initiator. An MSS terminates the execution of the second Echo protocol when it has received answers from all its children (in the spanning tree) and has consequently "echoed" its sz variable to its parent. After such termination, an MSS stops to broadcast "count" messages when a new mobile hosts enters its cell. After the completion of the second Echo protocol, the initiator MSS knows the total number of mobile hosts in the network (stored in its local sz variable).
4. The initiator MSS broadcasts a "(size,sz)" message in its cell and then starts a third execution of the Echo protocol by sending a "(inform-tok,sz)" message to its neighbors. This third execution aims at informing all mobile hosts about the size of the network. An MSS, upon receiving a "(inform-tok,sz)" message, broadcasts a "(size,sz)" message in its cell and forwards a "(inform-tok,sz)" message to its neighbors to continue the execution of the Echo protocol. If an MSS receives a "join" message, it broadcasts again the "(size,sz)" message. After the completion of the third Echo protocol, all MSSs have broadcasted the size of the mobile network in their cells. Finally, the initiator MSS starts a fourth execution of the Echo protocol to inform the MSSs about the completion of the counting algorithm. After the completion of the fourth Echo protocol, an MSS stops to broadcast "size" messages when a new mobile host enters its cell.

The four executions of the Echo protocol imply that the algorithm requires $8|E|$ messages to be exchanged in the fixed part of the network yielding a total cost of $O(mC_w + |E|C_f) = O(mC_w + n^2C_f)$ which is better than the cost of the VTA algorithm.

The simulated implementations of the VTA and the TTA algorithms were done on the DSP tool. One of the major difficulties in the experimental setup was the modeling of the speed and the type of movement of a mobile host. When a mobile host moves fast, it will change many cells during the execution of any counting algorithm and thus increase the overhead of keeping the routing tables of MSSs updated. Since the description of the speed in terms of physics was rather difficult, the approach followed in [1.16] was to associate the speed of a host with the propagation delay of messages in the fixed part of the network. In these terms, a *slow* mobile host is a host which does not change cell for $O(D)$ time units (where D is the diameter of the fixed part of the network). A host which changes cell in time smaller than $O(D)$ is called a *fast* mobile host.

Five different topologies were considered in [1.16] with n (number of MSSs) ranging from 20 to 100, $|E|$ (number of edges in the fixed part of the network) ranging from 50 to 310, and diameter ranging from 5 to 21. The transmission delay in all links was unary in order to avoid the overhead of message delay in the protocol execution time. In all topologies, there were 10 mobile hosts in each cell of an MSS. At the beginning of the simulation the mobile hosts were left to move randomly in the network in order to take random positions before a counting algorithm starts. For the VTA algorithm, the virtual topology constructed was basically a list of mobile hosts where the host with identity i considered as its neighbors the hosts with identities $i-1$ and $i+1$.

Three parameters were measured in all experiments: (i) the number M_f of messages exchanged in the fixed part of the network in order to deliver messages to the mobile hosts; (ii) the number M_r of radio messages transmitted by the mobile hosts (excluding the "join" messages, since these are also used for network control and routing purposes); and (iii) the execution time of the protocol. The last two parameters express the *battery power* of a mobile host, since this power depends on the number of message transmissions made by the mobile host as well as on the time the host remains active (algorithm execution time).

The TTA algorithm was significantly better than VTA in any topology and for any measurement parameter considered both in battery consumption and in the load of the fixed network. For example, in topology 5 ($n = 100$, $|E| = 310$, $D = 21$) and for slow mobile hosts, M_f was 29350 for VTA and 2480 for TTA, M_r was 2000 for VTA and 1000 for TTA, while the execution time was 18152 for VTA and 171 for TTA. Very similar results hold for fast mobile hosts.

1.5.3 Basic Protocols in the ad-hoc model

A fundamental problem in the ad-hoc model is to send a piece of information from some sender host to another designated receiver host. This *basic communication* or *routing* problem is a highly non-trivial task in ad-hoc mobile networks for several reasons: (a) local connections are temporary and may change as users move; (b) the movement rate of each user might vary, while certain hosts may even stop in order to execute location-oriented tasks.

The most common way to establish communication is to form paths of intermediate nodes (i.e., hosts), where it is assumed that there is a link between two nodes if the corresponding hosts lie within one another's transmission radius and hence can directly communicate with each other [1.14, 1.36, 1.43]. In other words, starting from the sender, each host broadcasts the message to all its neighbors until the intended receiver gets it (if possible). This protocol is called *flooding* and clearly requires a lot of messages. Indeed, this approach of exploiting pairwise communications is common in ad-hoc mobile networks that either cover a relatively small space (i.e., the temporary network has a small diameter with respect to the transmission range), or are dense (i.e., thousands of wireless nodes). Since almost all locations are occupied by some hosts, broadcasting can be efficiently accomplished.

In wider area ad-hoc networks however, broadcasting is impractical, as two distant hosts will not be reached by any broadcast since users do not occupy all intervening locations, that is, a sufficiently long communication path is difficult to establish. Even if such a path is established, single link failures happening when a small number of users that were part of the communication path move in a way such that they are no longer within the transmission range of each other, will make this path invalid. Note also that the path established in this way may be very long, even in the case of connecting nearby nodes.

A different approach to solve this basic communication problem is to take advantage of the mobile hosts natural movement by exchanging information whenever mobile hosts meet incidentally. Protocols based on this idea are divided into *non-compulsory* and *compulsory protocols.*

A *non-compulsory* protocol is one whose execution does not affect the movement of the mobile host. When the users of the network meet often and are spread in a geographical area, flooding the network will suffice. It is evident, however, that if the users are spread in remote areas and they do not move beyond these areas, then there is no way for information to reach them, unless the protocol takes care of such situations.

One way to alleviate these problems is to force mobile users to move according to a specific scheme in order to meet the protocol demands, thus yielding the so-called *compulsory* protocols. Such a protocol requests that *all* mobile hosts perform certain moves in order to guarantee correctness of the protocol.

A compromise between non-compulsory and compulsory protocols is introduced in [1.7]. The idea is to force only a small subset of mobile users, called the *support* Σ of the network, to move as per the needs of the protocol (the move of the rest is arbitrary and is not affected by the protocol). Such protocols are called *semi-compulsory* protocols. The support serves as an intermediate pool for receiving and delivering messages.

To address the crucial issue of modeling the motions of mobile hosts in the three-dimensional space, a rather general graph theoretic model was introduced in [1.17]. Under this model, the space S of motions is mapped to a graph $G = (V, E)$ called the *motion graph.* The graph is constructed as follows. The space S is quantized in 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 of S. Two vertices are connected by an edge if their corresponding cubes are adjacent. Note that the number of vertices n of G approximates the ratio of the volume of S and the space occupied by the transmission range of a mobile host. Since edges represent the (at most 6) neighboring polyhedra of a cube, it follows that $|E| = O(n)$. The mobile hosts move along the vertices and edges of the motion graph G (note that the motion graph model neglects the detailed geometric characteristics of the motion). It is assumed that the hosts know in advance (for example, from the hardware) the type and the dimensions of the polyhedron that is used for the quantization of S in order to be able to determine whether they have covered enough distance to reach a new vertex of G.

In the rest of this section, we shall discuss two efficient semi-compulsory protocols for the basic communication problem developed and implemented in [1.7, 1.8], and which model motions of hosts using the motion graph. Although "hard" input instances (in the sense of Section 1.4) were not considered, the experiments were conducted on several interesting "pragmatic" inputs.

1.5.3.1 The Snake Protocol. The first semi-compulsory protocol for the basic communication problem was presented in [1.7]. It uses a snake-like sequence of k support stations (i.e., they form a list of k nodes) that always remain pairwise adjacent and move in a way determined by the snake's head. As a consequence, the protocol is referred to as the *snake protocol.* There is a set-up phase of the ad-hoc network, during which a predefined number, k, of hosts, become the nodes or members of the support. The head is determined by performing a leader election between the members of Σ. Once determined, the head (denoted by M_0) assigns unique names to the rest of the support members $M_1, M_2, \ldots, M_{k-1}$. The motion of the support stations is accomplished in a distributed way via a support motion subprotocol P_1 which enforces the support to move as a "snake", with the head M_0 doing a random walk on the motion graph and each of the other nodes M_i executing the simple protocol "move where M_{i-1} was before". When some node of the support is within the communication range of a sender, an underlying sensor subprotocol P_2 notifies the sender that it may send its message(s). The messages are then stored in every node of the support using a synchronization subprotocol P_3. When a receiver comes within the communication range of a node of the support, the receiver is notified that a message is "waiting" for him and the message is then forwarded to the receiver. Duplicate copies of the message are then removed from the other members of the support. In this protocol, the support Σ plays the role of a (moving) backbone subnetwork (of a "fixed" structure, guaranteed by the motion subprotocol P_1), through which all communication is routed.

The snake protocol is theoretically analyzed in [1.7], where it is shown that the total expected communication or delay time to send a message from a sender to a receiver is at most $\frac{2}{\lambda_2(G)}\Theta(n/k) + \Theta(k)$ where G is the motion graph, $\lambda_2(G)$ is its second eigenvalue, n is the number of vertices in G, and $k = |\Sigma|$.

A first implementation of the protocol was developed and experimentally evaluated in [1.7] with the emphasis to confirm the theoretical analysis, and to investigate whether it is helpful for the head of Σ to remember past positions occupied by Σ, thus avoiding them in the future. The implementation was done in C++ using LEDA [1.33].

The experimental setup in [1.7] consisted of three kinds of inputs, one random and two structured ones. Each kind of input corresponded to a different type of motion graph. The motion graphs considered were random graphs (a natural starting point), 2D grid graphs (the simplest model of motion when mobile hosts move on a plane surface), and bipartite multi-stage graphs. The latter type of graph consists of a number s of stages (or levels) where each stage consists of n/s vertices. There are edges between vertices of consecutive stages chosen randomly among all possible edges between the two stages. This type of graphs is interesting as such graphs model movements of hosts that have to pass through certain places or regions, and have a different second eigenvalue than grid and random graphs (their second eigenvalue lies between that of grid and random graphs).

For all these types of graphs several values for n in the range $[100, 6400]$ were considered and different values for the support size k in the range $[5, 40]$. For each motion graph constructed, $1,000$ users (mobile hosts not belonging to Σ) were injected at random positions that generated 100 transaction message exchanges of 1 packet each by randomly picking different destinations (i.e., a total of $100,000$ messages were transmitted). The move of each user was random and independent of the protocol. Each experiment was carried out until all $100,000$ messages were delivered to the designated receivers. The synchronization subprotocol P_3 (storing every message to each member of Σ) was not implemented and hence the extra delay imposed by this subprotocol was not counted in the measured delay times. This does not affect the behaviour of the snake protocol and helps simplifying the implementation.

The conducted experiments [1.7] indeed confirmed the theoretical analysis that only a small support is needed for efficient communication, and indicated that limited memory (remembering just a few past positions) incurs a slight improvement on the delay time, while bigger memory is not helpful at all.

1.5.3.2 The Runners Protocol. The second semi-compulsory protocol for the basic communication problem has been recently presented in [1.8]. This protocol is based on the idea that the members of Σ do not to move in a snake-like fashion, but they perform *independent* random walks on the motion graph G, that is, the members of Σ can be viewed as "runners" running on G. In other words, instead of maintaining at all times pairwise adjacency between members of Σ, all hosts sweep the area by moving independently from each other. Consequently, this protocol is referred to as the *runners protocol.* When two runners meet, they exchange any information given to them by senders encountered using a new synchronization subprotocol P_3'. As in the snake case, when some node of the support is within the communication range of a sender, the underlying sensor subprotocol P_2 notifies the sender that it may send its message(s). When a user comes within the communication range of a node of the support which has a message for the designated receiver, the waiting messages are forwarded to the receiver. The runners protocol does not use the idea of a (moving) backbone subnetwork as no motion subprotocol P_1 is used. However, all communication is still routed through the support Σ and it is expected that the size k of the support (number of runners) will affect performance in a more efficient way than that of the snake approach. This expectation stems from the fact that each host will meet each other in parallel, accelerating the spread of information (that is, the messages to be delivered). A member of the support stores all undelivered messages in a set S_1, and maintains a list of receipts S_2 to be given to the originating senders. When two runners meet at the same site of the motion graph G, the synchronization subprotocol P'_3 is activated. The subprotocol imposes that when runners meet on the same site, their sets S_1 and S_2 are synchronized. In this way, a message delivered by some runner will be removed from the set S_1 of the rest of runners encountered, and similarly delivery receipts already given will be discarded from the set S_2 of the rest of runners. The synchronization subprotocol P'_3 is partially based on the *two-phase commit* algorithm as presented in [1.30].

The runners protocol turns out to be more robust than the snake protocol. The latter is resilient only to one fault (one faulty member of Σ), while the former is resilient to t faults for any $0 < t < k$.

In [1.8], a comparative experimental study of the snake and the runners protocols was conducted based on a new generic framework developed to implement protocols for mobile computing which constitutes part of the basic primitives provided by the Distributed Algorithms Platform [1.10] for wireless computing. Under this framework, the implementation of the runners protocol and the re-implementation of the snake protocol were carried out. All implementations were done in `C++` using LEDA [1.33] and the primitives of DAP [1.10]. To provide a fair comparison between the two different protocols (snake and runners), the subprotocol P_3 of the snake protocol was implemented in [1.8], and consequently the extra delay imposed by the synchronization of the mobile support hosts was also counted.

The experimental setup in [1.7] has been extended in [1.8] to include more pragmatic test inputs regarding motion graphs. Hence, except for the test inputs considered in [1.7] (random graphs, 2D graphs, bipartite multi-stage graphs; see Section 1.5.3.1), two other structured families were considered: 3D graphs (modeling 3D space), and two-level graphs. The latter class consists of dense subgraphs interconnected by a small number of paths. It was motivated by the fact that most mobile users usually travel along favourite routes (e.g., going from home to work and back) that usually comprise a small portion of the whole area covered by the network (e.g., urban highways, ring roads, metro lines), and that in more congested areas there is a high volume of user traffic (e.g., city centers, airport hubs, tourist attractions). In the conducted experimental study, the primary interest was to provide measures on communication times (especially average message delay), message delivery rate, and support utilization (total number of messages contained in all members of the support).

The experiments in [1.8] revealed that: (i) for both protocols only a small support is required for efficient communication; (ii) the runners protocol outperformed the snake protocol in almost all types of inputs considered. More precisely, the runners protocol achieve a better average message delay in all test inputs considered, except for the case of random graphs with a small support size. The runners protocol achieves a higher delivery rate of messages right from the beginning, while the snake protocol requires some period of time until its delivery rate stabilizes to a value that is always smaller than that of runners. Finally, the runners protocol utilizes more efficiently the available resources as far as memory limitations are concerned, as it has smaller requirements for the size of local memory per member of the support.

1.6 Modeling Attacks in Networks: A Useful Interplay between Theory and Practice

A recent thread of research concerns attacks in computer networks which pose several key problems regarding intrusion propagation and detection. Various models have been proposed under which researchers mainly study the effective detection and defeat of attacks assuming a very powerful intruder; see for example, [1.26, 1.35]. In this setting, intrusion propagation (the process of spread of such attacks) has mostly been investigated under gossip or epidemiological models [1.26]. On the other hand, the fear of malicious attacks along with the development of advanced cryptographic techniques has considerably increased the security level of current computer systems. Hence, contrary to previous models and approaches, a recent work [1.34] is concerned with studying intrusion propagation assuming that the intruder has a rather limited power and aims at investigating how intrusion can propagate in a perhaps highly secure network. To this end, a general model for such an intrusion and its propagation in networks is introduced. In the rest of this section, we shall discuss the particular model as well as the development of distributed protocols for attack propagation under this model. The interesting issue is that a tight combination of analytic and experimental methods is used to develop the protocols.

In the model introduced in [1.34], a network $\mathcal{N}$ is viewed as a collection of n host systems (nodes) each one having its own logical address. There is some underlying physical infrastructure whose specific topology is not a concern of the model. Communication is not necessarily done point-to-point. Direct communication between two nodes is achieved by establishing a virtual channel through the physical infrastructure between these two nodes.

Assume that in such a network an intruder, starting from his own computer, would like to break as many other systems as possible. The intrusion consists of a collection of attacks. An *attack* is issued from some node in $\mathcal{N}$ and is an attempt to break the perimetric security of another node (host system) in $\mathcal{N}$. The intrusion is realized by an attack scheme. An *attack scheme* is a protocol for the organization of the attacks issued from specific nodes of $\mathcal{N}$. The intruder is a greedy one, i.e., does not have a specific target, and attacks computer systems equiprobably at random. An attack succeeds or fails, independently of other attacks, with a failure probability $0 < f < 1$ that represents the difficulty of breaking a system in $\mathcal{N}$; f is a gross measure of the security level of the attacked systems (e.g., of the average security or the perceived maximum security level of a system) and may also depend on the intruder's skills. The model assumption about f is motivated by a large class of existing attacks; for example, attacks that are based on randomly sampling a set of possible passwords from a large password domain and then trying each of them. The probability of success of such a scheme in a node does not depend on previous successes at other nodes or on previous attempts at the same node. This is because the locally implemented set of passwords is perhaps different in each node and the set of passwords used by the local attack software is very small (for reasons of speed) compared to the password domain set.

If an attack does not fail, then some, randomly and equiprobably chosen, network node is returned. Because of that, it may happen that an already selected node (an already broken system) is chosen again. If the result of a non-failed attack is a node which has not been selected before, then the attack is considered *successful* and a *virtual link* (virtual channel) is established to that node. The random selection, with possible repetition, of a node in the case of a non-failed attack is motivated by the following pragmatic considerations: (i) if the local attack software (e.g., a worm) is successfully confronted, it should not reveal any information about broken nodes in the past; (ii) the local attack software may blindly extend attacks to hosts contained in tables of the newly broken systems which may include the already broken ones. The intruder tries to protect himself as much as possible from being traced: once a system is broken, his software tries from *that* system to attack (again equiprobably at random) another system by disguising itself as a user process of the broken system. Because of the danger to be discovered, the intruder's software can ever try only a limited (i.e., constant) number g of attacks from a specific node of the network. If a successful attack is issued before the limit g is reached, then the software enters a dormant phase and performs no action (for the purpose of not raising any suspicions). If at some node i the software exhausts the attack bound g, then it terminates execution at i and "backtracks" to a previously broken system j to continue from there its attacks, provided that there are still some attempts left at j. In such a case, the local software at j is reactivated and starts again to issue attacks. If at any time during the execution of the attack scheme, the intrusion is discovered by some system, it is assumed that the whole attack scheme to $\mathcal{N}$ terminates.

Two natural questions raised here are: (a) how long the intruder can go on (i.e., how many computer systems can be successfully attacked) in $\mathcal{N}$ until he is discovered, and (b) how many virtual links a detection mechanism has to trace in order to find the origin of the intrusion. In particular, assume that the intrusion starts at time 0 with attack scheme S. At any time $t \geq 0$, let $n_S(t)$ be the number of nodes captured, called the *spread factor*, and let $\ell_S(t)$ be the shortest possible distance (in number of virtual links) from the currently active position of the intruder's software to the origin, called the *traceability factor*. Given a discovery (i.e., stopping) time T, the problem is to estimate $n_S(T)$ and $\ell_S(T)$. This is referred to as the *attack propagation* problem. The goal, from the side of the intruder, is to employ an attack scheme which maximizes *both* factors. Note that this is a non-trivial task; for example, almost all epidemiological (and gossip propagation) models have usually very small $\ell_S(T)$ compared to $n_S(T)$, because of their "radially" spreading nature.

The above process defines naturally a graph G whose vertices correspond to the nodes of the network and if a virtual link (i.e., a successful attack through some virtual channel) is established between two nodes i and j, then an edge between vertices i and j is added to G. In this setting, the spread factor $n_S(T)$ is the size of the obtained connected component in G, and the traceability factor $\ell_S(T)$ is the length (number of edges) of the path in this connected component from the current vertex (node) issuing attacks to the origin (the node from which the intrusion was started). This path is referred to as the *traceability path*. Hence, the attack propagation problem reduces in estimating the values of these two quantities in G.

Another interesting issue is to investigate the possibility of a total failure of an attack scheme, namely the possibility that it eventually returns to its starting point, not because the intruder is discovered but due to backtracking caused by the limited number of attempts from a specific node.

In [1.34], the attack propagation problem is tackled by presenting four different protocols (attack schemes) by which an intruder can organize his attacks in the above model. The starting point is an attack propagation protocol that organizes attacks along a single traceability path. The g attacks per node are grouped into three equally sized sets. The first two sets (called *green* and *red*, resp.) are used to propagate the attack, while the third set is kept for restarting the attack scheme in case of a total failure. The protocol tries initially to establish a (long) traceability path, link by link, using only the green attacks. Each attack is issued from the last node in the path which is considered active (i.e., it possesses a token). An attack is considered successful if a new node is returned which will now become the last node of the path and gets the token. When attack propagation, that is, extension of the constructed path using green attacks, is not possible, then the red set of attacks is used.

If extension to a new node is established, then the protocol passes the token to that node and continues from there using its green attacks. Otherwise, it backtracks to the first node whose red attacks have not been used yet and (after passing the token) tries to extend the path from that node using the red attacks. The node having the token always stores the maximum (in length) traceability path of attacked systems constructed so far by the protocol. When the path shrinks to a single node, then that node notifies all nodes of the maximum traceability path seen in the past to try to use their third batch of attacks in order to restart the protocol. The above protocol is referred to as the *original protocol* and forms the basis for the development of three other protocols, called *tree protocols*.

The tree protocols are based on the fact that the graph G, constructed incrementally during the execution of the original protocol, is actually a tree (only successful attacks are recorded as edges of G). Hence, instead of keeping only the maximum traceability path constructed, the idea is to store the whole tree. Subsequently, various orders of the nodes in this tree for path extension are considered using their third batch of attacks. The different orders specify the different ways the intruder can use to organize his attacks. Clearly, the maximum traceability path constructed by the original protocol is the path of maximum depth in the tree. Hence, the tree protocols provide naturally a bigger front for expansion. The tree protocols differ in the order the nodes of the tree are considered for path expansion. Two protocols are based on "reverse" DFS, while the third one is based on "reverse" BFS.

The above protocols are theoretically analyzed in [1.34] where also an implementation of the protocols was carried out (in a simulation environment) along with a comparative experimental study. The development of the protocols constitutes an interesting case of distributed algorithm engineering. For the analysis of the protocols, both analytic and experimental methods were used that are actually tied to each other. The analytic study of the protocols, where applicable, was rather complicated and gave only lower bounds that are (probably) not tight. Hence, resorting to experiments was the only way to get insight as well as a basis of reasonable assumptions to further proceed with the analysis. For example, it was crucial in the analysis of the tree protocols to find a lower bound on the ratio between the traceability factor and the size of the tree. The experiments clearly demonstrated a lower bound of $1/2$ for this ratio which, along with the tree evolution observed experimentally, helped to analytically prove it and complete the analysis. Moreover, the implementation and experimentation with the original protocol provided useful feedback which was crucial in the development of the tree protocols.

The analytic and experimental methods in [1.34] show that for *any* $0 < f < 1$, there exists a g for which any of the above attack schemes will achieve a $\Theta(n)$ spread factor with high probability, provided T is sufficiently large. This means that if an intrusion is realized by any of the attack schemes, it will spread, regardless of the security level, to a big part of the network. It is also shown that the spread and the traceability factors are linearly related. Actually, for the tree protocols this linear relationship holds, with high probability, during the *whole* duration of the attack propagation. This implies that it will not be easy for a detection mechanism to trace the origin of the intruder, since at any time it will have to trace a number of links proportional to the number of nodes captured. Finally, it is shown that the probability of a total failure of any attack scheme is very small. The experiments conducted in [1.34] verified the theoretical results and exhibited the robustness of one tree protocol.

1.7 Conclusion

Distributed algorithm engineering has certain characteristics which are very different from conventional algorithm engineering. In this work, we made a first attempt to address these issues and suggest possible approaches that could efficiently tackle them. We hope that the suggested approaches will inspire researchers to further investigate these issues and result in more systematic methods.

References

1.1 A. Acharya, B.Badrinath, and T. Imielinski. Structuring Distributed Algorithms for Mobile Hosts. In *Proc. 14th International Conference on Distributed Computing Systems*, 1994.
1.2 M. Adler, H. Räcke, N. Sivadasan, C. Sohler, and B. Vöcking. Randomized Pursuit-Evasion in Graphs. In *Automata, Languages, and Programming* – ICALP 2002, *Lecture Notes in Computer Science* (Springer-Verlag, 2002), to appear.
1.3 H. Attiya and J. Welch. *Distributed Computing*. McGraw-Hill, 1998.
1.4 B. Awerbuch and D. Peleg. Concurrent Online Tracking of Mobile Users. *Journal of the ACM*, 42(5):1021–1058, 1995.
1.5 M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–863, 1984.
1.6 J. Buss, P. Kanellakis, P. Ragde, and A. Shvartsman. Parallel Algorithms with Processor Failures and Delays. *Journal of Algorithms*, 20:45–86, 1996.
1.7 I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis. Analysis and Experimental Evaluation of an Innovative and Efficient Routing Approach for Ad-hoc Mobile Networks. In *Proc. 4th Workshop on Algorithmic Engineering* – WAE 2000, *Lecture Notes in Computer Science*, Vol. 1982 (Springer-Verlag, 2000), pp.99-110.
1.8 I. Chatzigiannakis, S. Nikoletseas, N. Paspallis, P. Spirakis, and C. Zaroliagis. An Experimental Study of Basic Communication Protocols in Ad-Hoc Mobile Networks. In *Proc. 5th Workshop on Algorithmic Engineering* – WAE 2001, *Lecture Notes in Computer Science*, Vol. 2141 (Springer-Verlag, 2001), pp.159-171.
1.9 O. Dahl, K. Nygaard. SIMULA — an Algol-based Simulation Language. *Communications of the ACM*, 9(9):671-678, 1966.

1.10 Distributed Algorithms Platform. `http://ru1.cti.gr/~LEP-DAP/`.
1.11 Distributed Systems Platform. `http://helios.cti.gr/alcom-it/dsp`.
1.12 The Esterel language. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.
1.13 M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
1.14 Z. Haas and M. Pearlman. The performance of a new routing protocol for the reconfigurable wireless networks. In *Proc. ICC'98*, 1998.
1.15 Hatzis, G. Pentaris, P. Spirakis, and V. Tampakas. Implementation and Testing Eavesdropper Protocols using the DSP Tool. In *Proc. 2nd Workshop on Algorithmic Engineering* – WAE'98 (Ed. K. Mehlhorn), pp.74–85, 1998.
1.16 Hatzis, G. Pentaris, P. Spirakis, and V. Tampakas. Counting in Mobile Networks: Theory and Experimentation. In *Proc. 3rd Workshop on Algorithmic Engineering* – WAE'99, *Lecture Notes in Computer Science*, Vol. 1668 (Springer-Verlag, 1999), pp.95–109.
1.17 K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas, and R. Tan. Fundamental Control Algorithms in Mobile Networks. In *Proc. 11th Annual Symposium on Parallel Algorithms and Architectures* – SPAA'99, pp.251–260, 1999.
1.18 M. Herlihy and S. Rajsbaum. Algebraic Topology and Distributed Computing: A Primer. In *Computer Science Today: Recent Trends and Developments*, Ed. Jan van Leeuwen, *Lecture Notes in Computer Science*, Vol. 1000 (Springer-Verlag, 1995), pp. 203–217.
1.19 T. Imielinski and H. F. Korth. *Mobile Computing*. Kluwer Academic Publishers, 1996.
1.20 The IOA homepage. `http://theory.lcs.mit.edu/tds/ioa.html`.
1.21 J. R. Jump. *NETSIM Reference Manual, Version 1.0.* Rice University, 1993.
1.22 J. R. Jump. *YACSIM Reference Manual, Version 2.1.* Rice University, 1993.
1.23 P. Kanellakis and A. Shvartsman. *Fault-Tolerant Parallel Computation.* Kluwer Academic Publishers, 1997.
1.24 Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining Tentative and Definite Executions for Dependable Parallel Computing. In *Proc. 23rd ACM Symp. on Theory of Computing* – STOC'91 (ACM Press, 1991), pp.381–390.
1.25 Z. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proc. 22nd ACM Symp. on Theory of Computing* – STOC'90, (ACM Press, 1990), pp.138–148.
1.26 J. Kephart and S. White. Directed-Graph Epidemiological Models of Computer Viruses. IBM Research Report; also, in *Proc. IEEE Symp. on Security and Privacy*, 1991.
1.27 E. Koutsoupias and C. Papadimitriou. Worst-Case Equilibria. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science* – STACS'99, *Lecture Notes in Computer Science*, Vol. 1563 (Springer-Verlag, 1999), pp. 404–413.
1.28 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558-564, 1978.
1.29 M. C. Little and D. McCue. Construction and Use of a Simulation Package in C++. *C User's Journal*, 12(3), 1994. Also at `http://cxxsim.ncl.ac.uk/`.
1.30 N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.
1.31 M. Mavronicolas and P. Spirakis. The Price of Selfish Routing. In *Proc. 33rd ACM Symp. on Theory of Computing* – STOC 2001 (ACM Press, 2001), pp.510-519.
1.32 S. McCanne and S. Floyd. *ns Network Simulator.* `http://www.isi.edu/nsnam/ns/`.
1.33 K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

1.34 S. Nikoletseas, G. Prasinos, P. Spirakis, and C. Zaroliagis. Attack Propagation in Networks. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures* – SPAA 2001 (ACM Press, 2001), pp.67-76. To appear in *Theory of Computing Systems.*
1.35 R. Ostrovsky and M. Yung. How to Withstand Mobile Virus Attacks. In *Proc. 10th ACM Symp. on Principles of Distributed Computing* – PODC'91 (ACM Press, 1991), pp.51-59.
1.36 V. Park and M. Corson. Temporally-ordered routing algorithms (TORA): version 1 - functional specification. IETF, Internet Draft, draft-ietf-manet-tora-spec-02.txt, Oct. 1999.
1.37 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM,* 27(2):228-234, 1980.
1.38 C.E. Perkins. *Ad-Hoc Networking.* Addisson-Wesley, 2001.
1.39 P. Spirakis and B. Tampakas. Distributed Pursuit-Evasion: Some Aspects of Privacy and Security in Distributed Computing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing* – PODC'94 (ACM Press, 1994), short paper.
1.40 P. Spirakis, B. Tampakas, and H. Antonopoulou. Distributed Protocols against Mobile Eavesdroppers. In *Proc. 9th Int'l Workshop on Distributed Algorithms* – WDAG'95, *Lecture Notes in Computer Science*, Vol. 972 (Springer-Verlag, 1995), pp.160–167.
1.41 G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.
1.42 B. Weiss, G. Gridling, U. Schmid, and K. Schossmaier. The SimUTC Fault-Tolerant Distributed Systems Simulation Toolkit. In *Proc. 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* – MASCOTS'99, (IEEE Press, 1999).
1.43 Y. Zhang and W. Lee. Intrusion Detection in Wireless Ad-hoc Networks. In *Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing* – MOBICOM'00 (ACM Press, 2000), pp.275-283.