Designing a Queue Events System for road traffic simulator

Camelia Avram*, Adina Morariu*, Honoriu Vălean*, Adina Aştilean*

**Technical University of Cluj Napoca, Romania*
(e-mail:{camelia.avram, adina.pop, honoriu.valean, adina.astilean}@aut.utcluj.ro)

Abstract: The urban traffic is frequently perturbed by congestions, followed by usual delays, accidents and road closures that cause supplementary delays. It is important to use intelligent systems for traffic simulation, routing strategies and scheduling techniques in order to achieve an optimal usage of resources. Scheduling message queues has strong similarities with algorithms used for giving different service rates to flows within network routers. In this paper a message scheduler that uses time based credit scheduling is proposed, in which resources are partitioned among messages solvers by allocating credits. Various aspects of the overall system are investigated in particular in relation to throughput, delay, scalability with processors and performance isolation.

Keywords: computer simulation, concurrent programming, distributed control, communication protocols, computer communication networks.

1. INTRODUCTION

Nowadays researches in many domains depend on software simulations to model various processes or hypothetical scenarios that often cannot be satisfactorily expressed theoretically nor easily reproduced and observed empirically. Based on system identification an optimal simulation models are derived and implemented as distributed application (depending on the type of the process large scale architecture are taken into account). Depending on circumstances the time or event driven simulation is adopted. The event processing involves tasks and messages exchange (producing, handling and erasing) that imply choosing different scheduling techniques in order to achieve an optimal usage of resources.

Traffic congestion management was recognized as a major problem in modern urban areas, since it has caused much frustration and loss of people's time. Traffic congestion has been increasing worldwide as a result of increased population growth and changes in population density. Congestion reduces efficiency of transportation infrastructure and increase travel time, air pollution, and fuel consumption.

In this paper the authors propose a distributed architecture used to simulate large scale systems using virtual machines and modern communication techniques and tasks scheduling. Building discrete event simulators leverage modern virtual machines to achieve performance and quality of services.

Virtual machine-based simulation provides the transparency of the kernel based approach with the performance of a library based approach, using language-based techniques, but within a standard language and its runtime Barr et al. (2004).

The urban traffic is frequently perturbed by congestions, followed by usual delays, accidents and road closures that cause supplementary delays. The architecture of Urban Traffic Advisory System (UDAS) is described by Aştilean et al. (2002) and according to this paper several different clients can access the system to receive data about the road traffic. To extend the covered area (for simulation and routing strategies) the UDAS is implemented in a distributed architecture located on different servers.

The scheduling of message queues has strong similarities with algorithms used for giving different service rates to flows within network routers. For example, the Weighted Fair Share (WFS) queuing discipline, Demers et al. 1989, allows different service rates to be allocated to different queues according to some requested share. WFS were developed in the context of packet-switched networks to protect one network flow from another. It belongs to a set of virtual-time-based algorithms that provide fairness and/or delay bounds (Michael et al. 1996 and Sundell et al. 2003). Message scheduling is distinct from packet scheduling in that the time it takes for a message to be serviced cannot be easily estimated; in a packet scheduler, it is a simple function of the packet length. This means that the guarantees given by the message scheduler are by nature less precise according to Huang (1991), Hunt et al. (1994) and Shavit et al. (1999).

Most of the complex real world problems are solved using distributed environments (Gomez-Sanz et al. 2002 and Jennings 2001).

Authors propose a message scheduler which uses time-based credit-scheduling, in which resources are partitioned among messages solvers (MS) by allocating credits. The total credits allocated to a MS is a fraction (defined by the *share* given to the task) of the time period

T. The scheduler always chooses to schedule the MS with the highest number of remaining credits that has a non-empty queue, i.e. the scheduler is work-conserving. Each time a MS is scheduled, the scheduler measures how long it runs before it completes. This amount is deducted from the total number of credits for that MS.

The key motivation behind of this research is to create a simulation system that can execute discrete event simulations efficiently (different traffic situations, control algorithms, designing new roads), yet achieve the transparency (implies the separation of efficiency from correctness) within a standard language and its runtime.

2. SYSTEM ARCHITECTURE

Efficient control of urban traffic requires the implementation of a sufficiently accurate model allowing prediction of the effects of various control actions (such as adaptation of red – green phases at different intersections) Michael et al. (1996). Given the size of the plant it is important that one can use a distributed implementation of this simulation model. The computational efficiency is also improved by using a heterogeneous model, where some parts of the network are represented by a macroscopic model, other parts by a microscopic model. Long road sections with fairly homogeneous traffic conditions are represented by time driven macroscopic models describing the evolution over time of flow, speed and density of vehicles in different locations. In other parts of the plant it is more efficient to use event driven microscopic models representing the times at which individual cars cross certain boundaries. This is the case for short sections of road and for intersections. Each component represents some randomness in the evolution of the plant. Also the problem of connecting time driven and event driven components to each other in a computationally efficient way is discussed and a solution is proposed. Traffic lights possess sensors to provide basic information relating to their immediate environment. This includes road and clock sensors, measuring the presence and density of traffic and providing the time of day to the traffic light.
In Figure 1 the system architecture is presented.

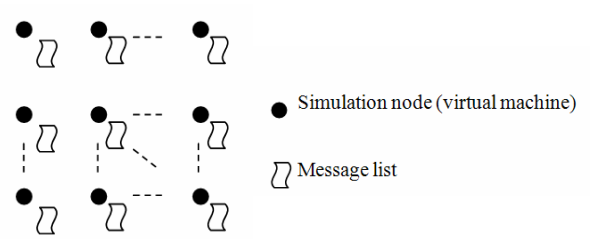


Fig. 1 System Architecture

The nodes are represented by computers or virtual machines (runs application servers) used to simulate the traffic behavior (collecting data, changing the control algorithms and different scenarios).

For data transfer among application servers a Queue Event System is designed, implemented and used. A distributed messaging system built in Java as the context is described further. Messages can be local messages (from the road traffic simulator, data acquisition, routing system etc.) or from a different PC (road traffic simulator, routing system) and have one receiver or several receivers (broadcast messages). A messaging system, where messages on different topics are carried from one or more producers to one or more receivers, with a discipline for time allocation, priorities rules and reallocation rules is designed and implemented in this work. The QES must work efficiently; in parallel (to accept several message producers); to avoid deadlocks and to use existent resources in an optimal way.

Messages can either be outgoing from local applications or in-coming from the network. In the first case, the associated task transmits them over the network; in the latter, it executes the associated application specific callback. The selection of the scheduling mechanism is determined by the needs of this messaging system. The arrival distribution of events and their dispatch time are not in general known *a priori*. As a messaging system may be used by different applications in different contexts, the means by which importance is attached to topics should be as general as possible. The scheduler must be work conserving and parallelizable, allowing it to make best use of the available resources and to scale with increasing numbers of processors. We don't assume any particular support from the underlying OS and assume that thread scheduling is non-preemptive.

Improvements to urban traffic congestion must focus on reducing internal bottlenecks to the network, rather than replacing the network itself.

Messages exchanged between simulation nodes have different length, priority and type and it is important to be processed using an algorithm which takes into consideration these requirements. Using the wireless communication is justified for the situation where the cabling connection is costly, inefficient and for the situation where the access to information is done during the way between different points with or without predefines position.

We used the wireless communication to collect information from traffic and for servers communication the wire connection is suitable due to the higher bandwidth and transfer rate offered.

A Queue Event System (QES) is used to manage the messages processing by each server.

The necessary data is collected from different sensors (installed on the road surface and counting the number of vehicles, cameras installed along the roads which provide visual information about the traffic, ultrasonic sensors etc.) locally stored and transmitted to other simulation nodes. The UDAS will provide information about the traffic such as: congestions, queues at the priority to all requests from the police, ambulances and fire department

and will also provide detailed information to the public transportation companies.

3. THE QUEUE EVENT SYSTEM

In the next section we motivate our choice of a time based credit scheduler for the messaging system and compare it with other alternatives. We then describe the implementation of the system showing how coordination between scheduling threads can be achieved using an appropriate lock free data structure. We give the invariant for this data structure and explain why this invariant is strong enough for use within the messaging system. We describe what fairness means for a work conserving scheduler on a multiprocessor system.

Messages can either be outgoing from local applications or incoming from the network. In the first case, the associated task transmits them over the network; in the latter, it executes the associated application-specific callback. The selection of the scheduling mechanism is determined by the needs of this messaging system. The arrival distribution of events and their dispatch time are not in general known a priori. As a messaging system may be used by different applications in different contexts, the means by which importance is attached to topics should be as general as possible. The scheduler must be work conserving and parallelizable, allowing it to make best use of the available resources and to scale with increasing numbers of processors.

The total amount of allocated credits to a Message Solver (MS) is a fraction of the ΔT time interval. The QES will give the access to the processor to the MS with the bigger credit. For each MS scheduled for execution QES determine the maximum credit time and this time is extracted from the total credit amount of all MS.

Let be $0 \le v_i \le 1$, v_i – is the credit allocated to MSi and s_i is the total number of messages from queue in a second, r_i – is the number of messages which are processed to the MSi in a second. The number of messages which are processed by MSi is formed by the number of messages guaranteed that can be processed (because of the allocated credit) and other messages which can be processed supplementary due to relocation of credits. R^{max} is the maximum rate, u_i – resource fraction which a MS try to consume, x_i ratio between v_i and u_i:

$$u_i = \frac{s_i}{\sum_j s_j} \quad (1)$$

$$x_i = \frac{v_i}{u_i} \quad (2)$$

For $x_i = 1$, MSi tries to use all the allocated credit, a value higher than 1 means that the MSi has more credit that is needed and a value smaller than 1 suggests that it is starving for more credit. Given value for v_j, s_j and R^{max}, r_i is:

$$r_i = Min\left(s_i, v_i * \frac{R^{max} - \sum_{x_j > x_i} r_j}{1 - \sum_{x_j > x_i} v_j}\right) \quad (3)$$

Algorithm for message scheduling:

```
Thread SchedulingThread
  while true do
    t→SchedulableTasks.get();
    if t.credit = 0
      then
      resetAllTaskCredits();
      /* the MS with bigger credit has 0. deltaT is
      over*/
        continue while
    end if
    e→t.eventQueue.get();
    To→getTime();
    t.processEvent(e);
    t.credit→t.credit − (getTime() − To);
    if --t.queueSize > 0
      then
      SchedulableTasks.put(t);
        /* the MS with more messages is put back for
        scheduling */
      end if
  end while
```

A MS, t, has an event queue t.eventQueue where the messages are introduced based on allocated credit, t.credit. A message is processed when MS has credit and the queue has at least one item.

We refer to the priority queue as *SchedulableTasks*. The MS with the most credit has the highest priority and is therefore at the head of the priority queue, from where it is removed by a scheduling thread for execution. Once invoked, the MS processes exactly one event. The scheduling thread measures the execution time and uses this to decrease the credit of the task, thus changing its priority. If the highest priority MS is out of credit, then no MS has credit left and the scheduling thread resets the credits of all MSs, i.e. the scheduler is work conserving. This situation occurs at the latest when the scheduling period *T* expires; but may occur earlier if some of the MS had less work than their allocated share. Using multiple scheduling threads allows the scheduler to run on multiple CPU cores. For efficient operation, it is important that a scheduling thread is not blocked by another.

All the information about the credits a MS has is maintained within the MS itself, and because it is only updated by the scheduler thread that took the MS out of *SchedulableTasks*, this information does not need to be thread-safe.

The event queue of tasks is implemented as a Michael Scott lock free FIFO queue. This queue allows scheduling threads (the readers) and input threads or application threads (the writers) to concurrently access the same queue without the need for locking.

Writer threads and scheduling threads coordinate in deciding whether a MS is schedulable or not. A writer thread that adds a new event to a MS that is currently not scheduled will add it to *SchedulableTasks*. Likewise, a scheduling thread that finds the event queue of a MS empty will remove the MS from the set.

The scheduler thread removes the MS from *Schedulable-Tasks* and returns it if there is still work to do (i.e., an event is in the task's queue). If the scheduler thread detects that there is no more work to do for a MS, it is not returned to the queue. It is the responsibility of the writer thread to put the MS back into *SchedulableTasks* when the MS has again events in its queue. This coordination is achieved without locking by using an atomic counter t.queueSize for the number of events in the MS's event queue.

The *writer thread* increments t.queueSize after having written an event, whereas the *scheduler thread* decrements it after reading an event. The scheduler thread will only return the MS to the set if the value is non-zero. A writer recognizes that a scheduler thread did not return a MS if the value before it succeeded in incrementing the atomic counter was zero.

We find that for time granularities that make sense in a messaging system running over a LAN (in the 100 – 1000 μs range) one level of messages groups is adequate.

Let be:
x – a message with higher priority than message y.

Get()A; Put(x)B; Ok()B; Put(y)B;

Ok()B; Ok(y)A; Get()C; Ok(x)C;

Each operation consists of a start of invocation and its completion. For example, *Put*(x) *A* stands for the start of the put invocation of message x into the priority queue by thread *A*, and *Ok*() *A* corresponds to its completion. The history is not linearizable when the priority of x is higher than the priority of y as no *legal* sequential history can respect the fact that the addition of x completed before the addition of y but was removed after. The practical consequence of this is that thread *A* retrieves a lower priority task than thread *C*, even though the get operation of thread *A* precedes the get of thread *C* and task x was put in before message y. Fig.2 illustrates this inversion of priorities.

The concurrent priority queue
Let a put operation be defined as follows:
PUT ::= [e : Element, start : Time, end : Time]

where *start* and *end* are the times that the operation started and completed at and *e* is the element added with priority *e.prio*. Let a get operation be defined as:

GET ::= [p : PUT, start : Time, end : Time, e : Element]

where *start* and *end* are the times that the operation started and completed at and *p* is the put operation whose element the get operation retrieves. It must be the case that an element is retrieved only after it has been added.

Vg : GET , g.end > g.p.start

We define a history of the observed get operations ordered by their time of completion.

H : SEQ of GET , 8i < j H[i].end _ H[j].end

Then the following must hold 8i, j i < j:
H[i].e.prio < H[j].e.prio => H[i].start < H[j].p.end

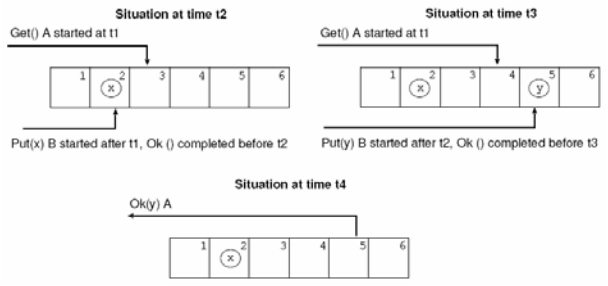


Fig.2. The put and get operation (t1<…<t4)

For example, suppose we have three tasks T_0, T_1 and T_2 with shares $v_0 = 0.6$, $v_1 = 0.1$ and $v_2 = 0.4$, suppose further that we know that $R^{max} = 10000$. We would like to know what values are received for the three messages if $s_0 = 4{,}000$, $s_1 = 6{,}000$ and $s_2 = 5{,}000$. It is clear that r_0 is 4,000 because message T_0 is sending less than its share. That leaves 6,000 to be shared among T_1 and T_2. Both T_1 and T_2 will get their guaranteed share (3,000 and 2,000 respectively) plus some fraction of the spare capacity (1,000) that T_0 has reserved but is not using. The unreserved capacity is shared proportionally to the shares v_1 and v_2, so T_1 gets 60% and T_2 gets 40%, meaning that $r_1 = 3{,}600$ and $r_2 = 2{,}400$. We now give the general formula for calculating r_i for an arbitrary resource allocation between a set of tasks and an arbitrary attempted sending rate on those tasks on a single processor.

In all of the tests the following configuration has been used unless otherwise stated. The time period over which the shares are valid is set to 10 ms; the number of messages used in the priority queue is set to 100; the communication between machines is always Megabit Ethernet, and TCP for the transport layer with the socket size set to 128 Kbytes is used.

To calculate the execution time for each contractor task, a model based on cellular automata is proposed.

A cellular automaton is a discrete model studied in computability theory, mathematics, theoretical biology and Microstructure Modeling. It consists of a regular grid of *cells*, each in one of a finite number of states. The grid can be in any finite number of dimensions. Time is also discrete, and the state of a cell at time *t* is a function of the states of a finite number of cells (called its *neighborhood*) at time $t - 1$. These neighbors are a selection of cells relative to the specified cell, and do not change. Every cell has the same rule for updating, based on the values in this neighborhood. Each time the rules are applied to the whole grid a new generation is created (Willox et al. 2003 and Martins et al. 2000).

In this case the cells represent the contractor task/thread. Each cell can have maxim of 8 neighbors, like in Fig.3.

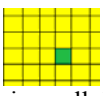


Fig.3 Contractor task using cellular automata

The cell stats are:

- 0 – for the active task (green colour);
- 1 – for the waiting task (yellow colour);
- 2 – for the task that had finalized their work (red colour);

The cells can change their states from 0 to 1, from 0 to 2 or from 1 to 0. A cell passes from 0 to 1 when the initial execution time is finished and the task is not accomplished. A cell passes from 0 to 2 when the initial execution time is finished and the task is accomplished. A cell passes from 1 to 0 when a new execution time is allocated for the task.

A simple example is presented in Fig.4. At the initial step (Fig.4.1) we have 9 active tasks. Each task has allocated by the QES an initial execution time "s_0". After the time "s" is finished, the cells have the states presented in Fig. 4.2. At the next step (Fig.4.3) all waiting task will be active, a time "s1", where "s1" is given by the QES according with the task priority.

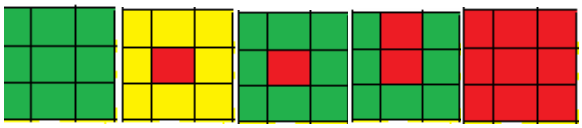


Fig.4.1. Fig.4.2 Fig.4.3 Fig.4.4 ... Fig.4.11

4. PERFORMANCE AND RESULTS

The scheduler guarantees that *when* a scheduler threads executes it will choose the *available* task that best fits the schedule. In case of a machine with one CPU in which the scheduler thread is the only thread that runs, then all tasks will always be available and the share allocated to the task will simply be a fraction of the CPU. In a real system here are many other threads running (messaging system, timer threads, monitoring threads of the I/O, threads supporting the control part of the messaging protocol and also JVM normally runs several threads). In conclusion the scheduler threads are not all the time running and the allocated share time for a task on a single machine is a share of a fraction of the time when the scheduler runs.

The problem becomes more complex in a multi-core architecture because when a scheduler thread is serving a task, that task is unavailable to other scheduler threads. This is simply a consequence of the fact that a task is allocated to at most one scheduler thread at any given moment.

The number of processors on which scheduler threads run in parallel is the product of the number of processors and their probability of getting scheduled. The maximum percentage of total scheduling time that any task can get is then given by (4):

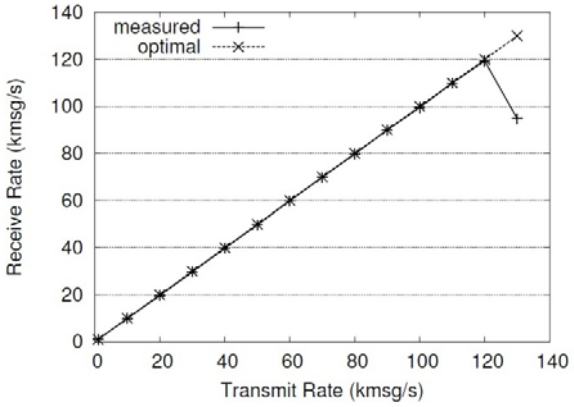
$$\frac{100\%}{Max(1,N*P)} \quad (4)$$

where: N – number of processors and P – represents the probability that a scheduler thread is scheduled by the OS. If P = 1, then no task can ever get more than 1/N of the total time the scheduler's threads run. For fixed N this approaches 100% as P gets smaller, and for fixed P it approaches 1/N as N gets bigger.

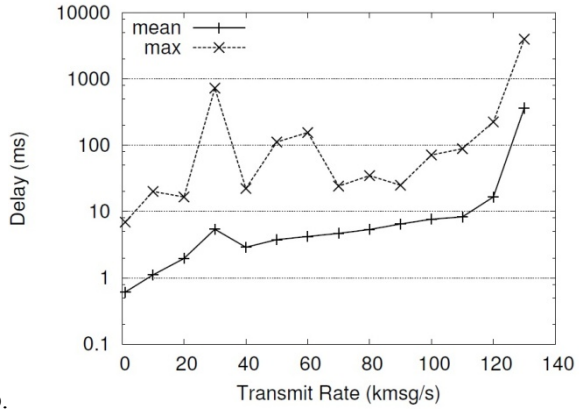
We run several tests for testing the proposed QES having the time period set to 10 ms. Number of different types of messages is 5 and for each core of the machine a scheduler thread is used. The communication between machines is a Megabit Ethernet, and we use TCP for the transport layer with the socket size set to 128 Kbytes. The message size is variable (can have 32 bytes, 64 bytes, 128 bytes length). All machines are running with Windows with Java 6. The machine on which the subscribers run has 2 × 2.3 GHz 2 core processors, i.e. 4 cores in total.

We measure the maximum number of messages per second we can send between a single publisher and a single subscriber (either PCs or virtual machines).

Fig.5 shows the evolution of the effective throughput and the end-to-end delay as a function of the publishing rate. The system sustains a rate of 120000 msg/s. The average end to end delay rises from below 1 ms for 1000 msg/s to slightly above 10 ms at 120000 msg/s. Above this figure the system is no longer sustainable, and the average delay rises dramatically.



a.



b.

Fig.5. Bandwidth/delay for one publisher to one subscriber.

5. CONCLUSIONS

In this paper authors have shown how a messaging system supporting different qualities of service for different

topics can be built using an event dispatching model; motivating the choice of time based credit scheduling within the messaging system and given a statement of the guarantees that it provides.

Finally, various aspects of the overall system are investigated in particular in relation to throughput, delay, scalability with processors and performance isolation.

ACKNOWLEDGMENT: This paper was supported by the project "Develop and support multidisciplinary postdoctoral programs in primordial technical areas of national strategy of the research - development - innovation" 4D-POSTDOC, contract nr. POSDRU/89/1.5 /S/52603, project co-funded from European Social Found through Sectorial Operational Program Human Resources 2007-2013.

REFERENCES

Aştilean, A., Avram, C., Leția, T., Hulea, M., and Vălean, H. (2002). Using Mobile Data Services for Urban Driving Advisory Systems, *Mobile Open Society through Wireless Telecommunications - MOST Conference*, Varsow, Poland, pag. 4, ISBN 83 - 87091 - 32 4.

Aştilean, A., Leția, T., Vălean, H., and Avram, C. (2004) Patients Monitor System Based on the Bluetooth Technology, *IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics*, pp. 403-408, ISBN 973-713-046-4, Romania.

Avram, C., and Boel, R. (2005). Distributed Implementation of A Heterogeneous Simulation Of Urban Road Traffic, *European Conference on Modeling and Simulation*, Riga.

Barr, R., Haas, Z.J., and van Renesse, R., JiST. (2004) An efficient approach to simulation using virtual machines, *Software – Practice and Experience, John Wiley & Sons, Ltd.*

Demers, A., Keshav, S., and Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm, *Symposium Proceedings on Communications Architectures & Protocols*. New York, NY, USA: ACM Press.

Huang, Q. (1991). An evaluation of concurrent priority queue algorithms, *Massachusetts Institute of Technology*, MIT Cambridge, MA, USA.

Hunt, G. C., Michael, M. M., Parthasarathy, S., and Scott, M. L. (1994). An efficient algorithm for concurrent priority queue heaps, *University of Rochester*, Rochester, NY.

Martins, M.L., Ceotto, G., Alves, S.G., Bufon, C.C.B., Silva, J.M., and Laranjeira, F.F. (2000). A Cellular Automata Model for Citrus Variagated Chlorosis, eprint arXiv: cond-mat/0008203.

Michael, M. M., and Scott, M. L. (1996). Simple, fast, and practical nonblocking and blocking concurrent queue algorithms, *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*.

Searle, J.R. (1969). Speech acts: an essay in the philosophy of language. *Cambridge University Press,* Cambridge, UK.

Shavit, N., and Zemach, A. (1999). Scalable concurrent priority queue algorithms, *PODC '99: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM Press.

Sundell, H., and Tsigas, P. (2003). Fast and lock-free concurrent priority queues for multi-thread systems, *Proceedings of the 17th International Parallel and Distributed Processing Symposium*. IEEE press.

Willox, R., Grammaticos, B., Carstea, A.S., and Ramani, A. (2003) Epidemic Dynamics: Discrete-Time and Cellular Automaton Models, *Physica*.