Parametric, Probabilistic, Timed Resource Discovery System

Camille Coti
LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité
camille.cot@lipn.univ-paris13.fr

This paper presents a fully distributed resource discovery and reservation system. Verification of such a system is important to ensure the execution of distributed applications on a set of resources in appropriate conditions. A semi-formal model for his system is presented using probabilistic timed automata. This model is timed, parametric and probabilistic, making it a challenge to the parameter synthesis community.

1 Introduction

The behavior of distributed systems in general can be challenging to prove. On the other hand, model checking techniques are particularly well adapted to verify that they follow a certain specification, because such techniques explore every possible execution of the system. Moreover, the execution of a distributed algorithm can depend on a large number of parameters, because of the complex combination of elements involved. It is therefore difficult to have a quantitative evaluation of what happens in the system for each and every value of each parameter.

This paper focuses on a resource management system. For some applications, resources can be shared between clients. These clients may need to access the resources in exclusive mode. As a consequence, a reservation system is necessary to arbitrate between clients and orchestrate the utilization of the resources.

For instance, a laboratory can buy a set of computation nodes and put them together in a cluster. A specific node, called the front-end, is used to access the computation nodes and a batch scheduler installed on this front-end node issues reservations on the nodes. The nodes cannot be accessed by a user that does not have an ongoing reservation on the said node issued by the batch scheduler. In practice, this reservation system is implemented using a single job queue that maintains a list of jobs that need to be executed and a list of available resources, and schedules the former on the latter [1, 9, 3, 2].

However, in many situations, this architecture cannot be implemented. For instance, a small lab who bought a handful of GPU nodes might not want to dedicate hardware and human time to setup and administrate a front-end node with a batch scheduling system. On some critical systems, this component can be seen as a single point of failure and then reduce the reliability of the whole system. However, in these situations, it is still necessary to make sure that applications have exclusive access to the machines they are using.

In this paper, we present a completely distributed reservation system, based on the state maintained by each machine itself, and the local network protocol Zeroconf [4]. This system was modeled using (timed) Petri Nets in [11, 10]. However, we have seen that this system is highly parametrizable. the purpose of this paper is to present it as a parametric system for verification and parameter synthesis. The resulting model has a high level of complexity, and therefore forms a case-study which cannot be reasonably solved with current methods. With this paper we aim at presenting it in order to open a discussion for methods that would tackle these problems, for instance by combining or hybriding methods.

T. Brihaye, B. Delahaye, L. Jezequel, N. Markey, J. Srba (Eds.):
Cassting Workshop on Games for the Synthesis of Complex Systems
and 3rd International Workshop on Synthesis of Complex Parameters
(Cassting'16/SynCoP'16).
EPTCS 220, 2016, pp. 53–62, doi:10.4204/EPTCS.220.5

The rest of this paper is organized as follow. The global architecture of the system and the algorithms are described in Section 2. Section 3 describes how it can be modeled. Section 4 specifies the expected behavior of the system and how parameter synthesis is useful to verify this behavior. Finally, section 5 concludes this paper and opens questions for the community.

2 Presentation of the system

This section presents the reservation system itself. The global architecture is described in section 2.1, the algorithms are presented in section 2.2 for the reservation protocol itself and in section 2.3 for the machines.

2.1 Architecture

The global architecture of the system, named QURD, is depicted in Figure 1. Computing resources *declare themselves* on the Zeroconf bus, and users/clients look on the Zeroconf bus to see which resources are available. An *application* (or a *job*) is made of several *processes* that are meant to run on a set of *resources*, also called *machines*. The user submits an application through a *client*.

The Zeroconf bus [4] is a network protocol used for self-configuration of network services. It was originally designed to allow automatic configuration of computers without any intervention from the user nor any centralized server, and later extended to various services. For instance, it is widely used for services such as DNS or network printers. In the latter example, printers declare themselves on the Zeroconf bus, and workstations look on the Zeroconf bus to find out which printers are available.

Zeroconf uses multicast UDP datagrams. It features three operations, two of them can be used for automatic service detection: *discover* and *advertise*. The third operation is called *resolution*: for instance, it is used by the multicast DNS protocol (mDNS) works as follows: 1) the client sends a multicast datagram to ask "what is the IP address that corresponds to this symbolic name"; 2) the hosts that has this symbolic name name answers with a unicast message to the client. A host that provides a service would typically *advertise* it. For this purpose, it sends a multicast datagram to tell all the machines of the networks "I provide this service and on that port". The other mode that can be used for service detection, *discover*, works the other way around: 1) a client sends a multicast datagram to ask "who provides this service?"; 2) servers that host the requested service reply to the client using a unicast datagram.

In our system, machines declare themselves on the Zeroconf bus when they are available and withdraw themselves when they are taken by a client, *i.e.* when a job is running on them: they use the *advertise* mode. Clients listen and receive the multicast declarations that are sent on the network.

However, this is not sufficient to ensure exclusive access to the machines, because of the asynchronous nature of the system and the absence of a consistent view between concurrent clients. Besides, the Zeroconf protocol does not have real-time accuracy: a machine which is not available anymore might be still visible on it. For instance, if a resource is available at a given moment, two clients will see it on the Zeroconf bus. Both will issue a request, but only one of them should get it.

2.2 Reservation algorithm

When a client wants to reserve a set of machines, the submission protocol works as follows. The client listens to the Zeroconf bus for available machines. It gets a list of available machines. It contacts them one by one and waits for a reply. If the machine acknowledges the reservation, the client receives an "OK" message and keeps the machine. Otherwise, the client receives a "KO" message.

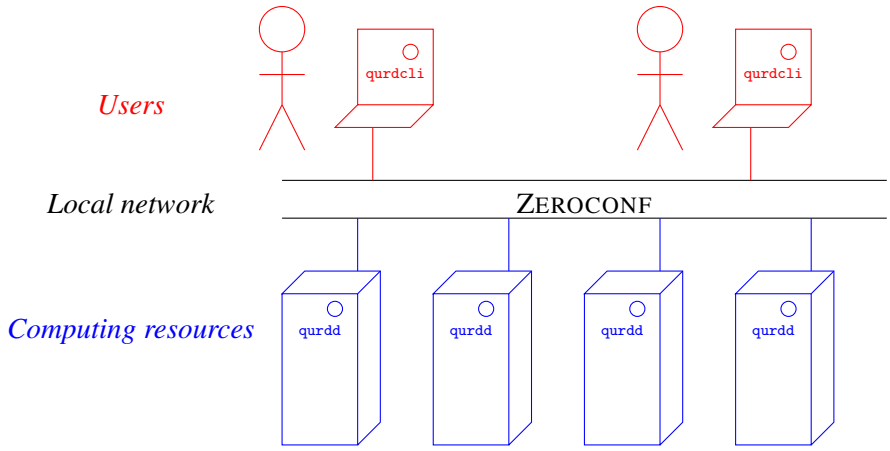

Figure 1: Architecture of a QURD system.

Once the client has enough machines, it starts its job on the said machines. Otherwise, two behaviors are possible. Either the client waits for other machines to become available, with a limit of time set by a timeout (called the *wait semantics*), or it cancels its reservation and frees the machine it has reserved, to try again later (called the *fail semantics*). The algorithm corresponding to the latter approach is given by algorithm 1.

Algorithm 1: Resource reservation algorithm (fail semantics)

```
reserveNodes( nbNodes) begin
    Data: machines = {}
    listenZeroconf();
    foreach machine m newly discovered do
        if card( machines ) < nbNodes then
            contactMachine( m ) ;
            ack = receiveAck( m );
            if ack == OK then
                machines.append( m ) ;
    if card( machines ) == nbNodes then
        return machines ;
    else
        freeMachines( machines ) ;
        return {} ;
```

One important issue with these two reservation semantics is to avoid deadlocks between concurrent reservation requests. For instance, if we consider a system with three machines and two concurrent reservation requests, one for two machines and one for three machines. If the first request gets one machine and the second gets two machines, neither of them has obtained enough machines and there is

no spare machine left. This situation would lead to a deadlock without the possibility to release and retry (*fail* semantics) or release everything after a timeout (*wait* semantics). The *wait* semantics is still useful if there is no available resource because some of them are used by running applications. In this case, as soon as an application is done, the request can get its resources.

2.3 Exclusive access protocol

In order to make sure that jobs have exclusive access to the machines that have been assigned to them, each machine implements a protocol based on its state. When a machine is available for jobs, its state is set to *available*. It can be reserved only when it is in the stat *available*. Otherwise, in any other state, it answers all the requests with "KO".

2.4 End of a job

Once the process running on a machine is done, the machine's state is set to *finished*. A job ends when all the processes are done: the client keeps track of which machine is done. Once the job is finished, the machines switch back into state *available* and republish themselves on the Zeroconf bus.

2.5 Resource volatility

Resources can be subject to failures. If they fail while they are idle, they switch to state *unavailable* or, if they are still visible on the Zeroconf bus, they never answer the clients' requests and the clients try to find another resource. If they fail while they are in state *reserved*, they never confirm the start-up of the application to the client it has been reserved by and the client handles this case too. If it fails when it is in state *finished*, the client has already taken into account the fact that this resource is done executing its part of the job, so it has no consequence on the execution of the process. The resource goes to state *unavailable* instead of *available*.

The main issue is when the failure happens when the resource is running the job. The failure is detected by a failure detector [6] and the machine is considered to be *dead*. The client is notified of this failure and tries to find a new machine to replace the failed one.

3 Modeling the system

This section presents models for the two parts of the system: the machines (section 3.1) and the reservation system of the clients (section 3.2). These two components interact with each other. These interactions are detailed in section 3.3.

The models are presented using a finite-state formalism. Transitions between states are represented by edges. Guards can be present on edges, given between brackets. When the system can chose between two transitions depending on a probability, the probability to chose a given edge is given between parenthesis. Edges related with each other by a probability are linked by an arc. The actions taken by a transition is given on the corresponding edge. In particular, when a transition can be taken only after T units of time (*i.e.* the system must remain in a state during at least T units of time), the time is initialized before entering the state by an action on the incoming edge ($time := 0$), and a guard on the outgoing edge sets the condition on the time to take this transition ($time > T$).

Interactions between automata are modeled by synchronizations on actions, in a similar way as what is presented in [8]. For instance, when an automaton sends a request, the corresponding edge contains the `request!` action and the automaton receives it with `request?` on the guard of an edge.

3.1 Model of each machine

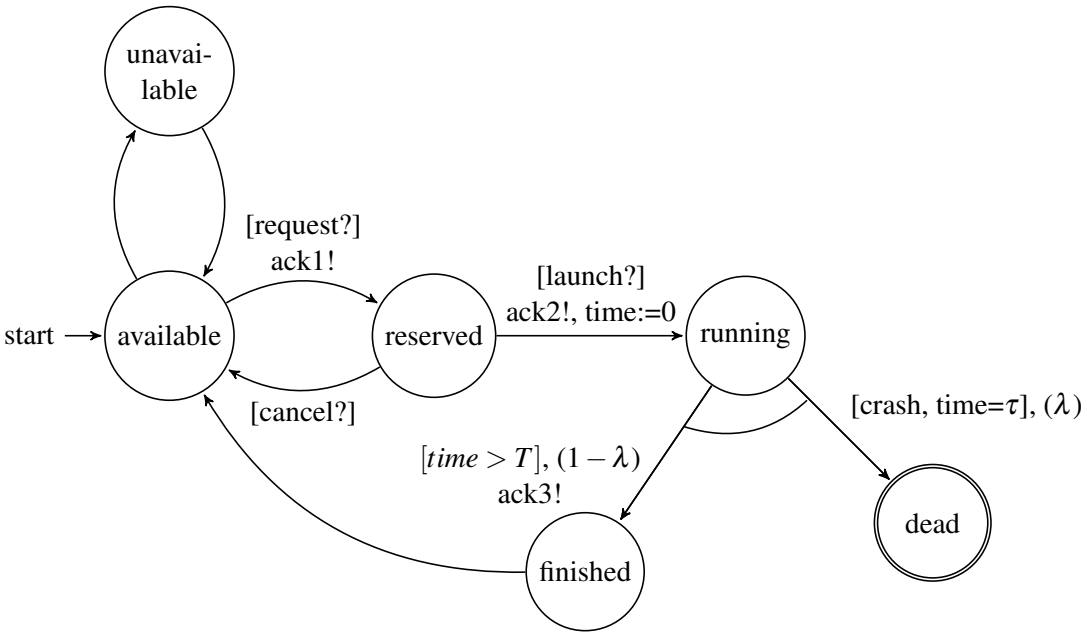

Figure 2: Automaton modeling the algorithm running on each machine.

The states of a machine are represented in the automaton depicted on figure 2. When no job is running on it, a machine is in the *available* state. For some (local) reason, such as an action from an administrator or a local user (when the machines are unused workstations, for instance with desktop grids), it can become *unavailable*. When the machine is in the state *available*, it can be reserved by a client: in this case, it enters the state *reserved*. We have seen in section 2.2 that a client can cancel a reservation; in this case, the machine returns to the state *available*.

Once the client has all the machines it needs to start a job, it sends a command to all of the machines it has reserved. These machines enter the state *running*, because it is running a process from a job. However, the machine can crash or fail during the execution. It happens with probability λ: then, the machine enters the state *dead*. The probability is given between parenthesis on the edges between states.

A more detailed model is given in figure 3. The resource has a probability λ of dying. If it dies, it reaches the state *fragile*. If it does not, if reaches the state *sustain*. It stays in the sustain state for (at least) T time units, representing the execution time of the process. Figure 2 is a more compact representation of this subpart of the model, since the transition after the state *running* is conditioned by a probability and a time.

Once the execution is done (*i.e.* after a given period of time and with probability $1-\lambda$), it enters the state *finished* and then can be *available* again.

When the machine is reserved, the machine notifies the client it acknowledges the request. Similarly, when the machine starts the application, it acknowledges the client. At the end of the execution, it notifies

the client it is done with its local process.

We can seen that a machine can evolve between states, except if a failure happens and it dies. In this case, it stays in the state *dead* until an administrator performs an action to fix the problem and restart the machine, which can take a long time compared to the typical execution time of each transition of the automata presented here. As a consequence, we are considering here that the machine stays in the state *dead*.

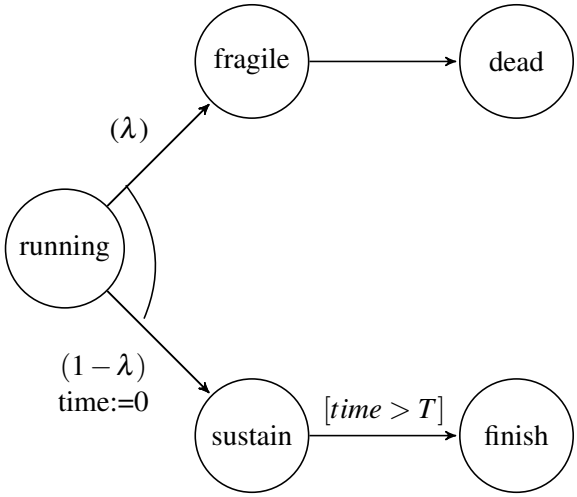

Figure 3: Modeling the volatility of a resource.

3.2 Model of the reservation system

The reservation system is modeled by the automaton given on figure 4. It uses counters. Initially, the system is in state *begin*. A counter is used to count the number of machines that have acknowledged the reservation; it is initialized to zero when the system transitions from the state *begin* to the state *reserve*. Each time a machine acknowledges the reservation, the counter is incremented. When the required number of machines have acknowledged (*NB OK* answers), the automaton reaches the state *launch*. In a similar way, it counts the number of machines that have acknowledged application start-up. Once they all have answered, the automaton reaches the state *wait*, until all the machines are done.

The automaton presented here implements the *wait* semantics. If not enough machines can be reserved, the system releases the machines and returns to the initial state.

A detailed model for the state *reserve* is given in Figure 5. This model is made of two parts. The top part of the model is the *discovery* system of Zeroconf: the client listens to the Zeroconf bus, and sends a request when it discovers a new machine. In the same time, the client waits for acknowledgements from the machines: this corresponds the lower part of the model. Every time an acknowledgement is received, a counted is incremented. When enough machines have answered, both automata reach the final state by synchronizing on the `done` action.

We have seen in section 2.5 that resources can fail during the execution of a job. In this case, the reservation system is informed by the failure detector and reaches the state *failure*. Then it requests an additional resource and starts the application on it.

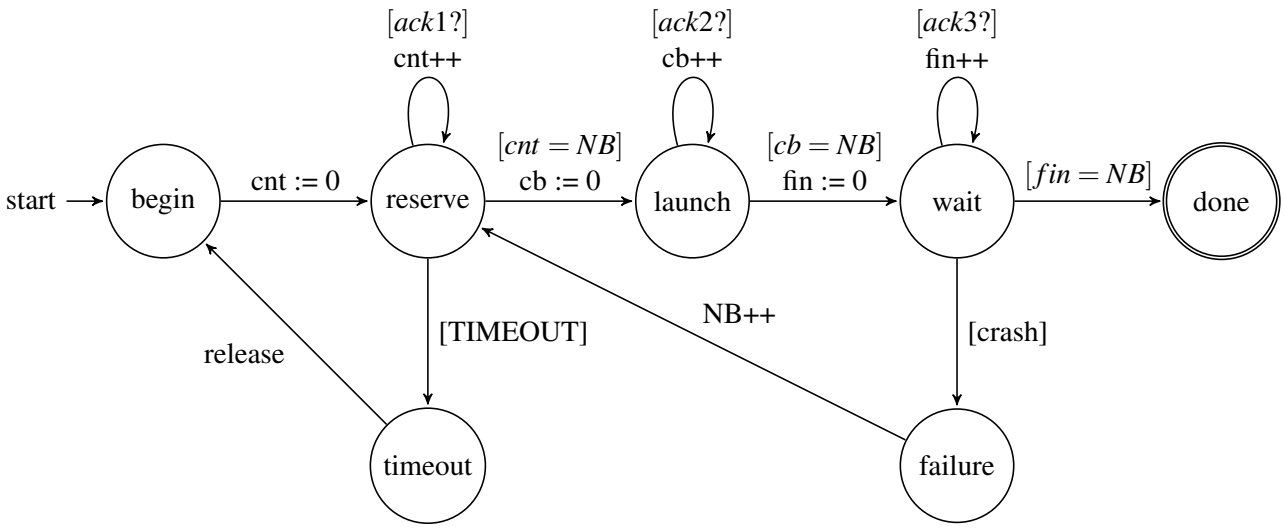

Figure 4: Automaton modeling the reservation system (*wait* semantics)

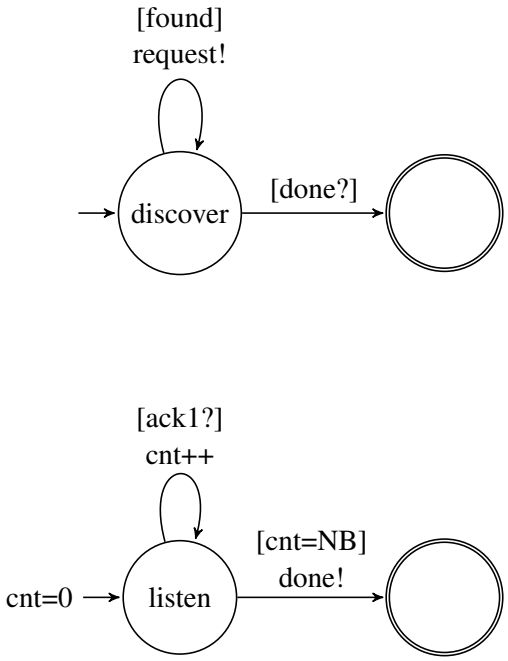

Figure 5: Detailed model for the state *reserve* (wait semantics)

3.3 Interactions between the automata

We have seen in Figure 2 that the machines need some actions from the reservation systems and that some actions are made to this reservation system. In a similar way, we have seen in figure 4 that the reservation system interacts with the machines. This set of actions forms a mini-protocol between the two automata.

1. The reservation system *sends a request* to the machines it has found on the Zeroconf bus (`request` action). When a machine receives a request, it answers *OK* or *KO* depending on the state it is currently in.
2. The available machines *acknowledge* the request (`ack1` action). For each acknowledgement it receives, the client increments a counter.
3. The reservation system *sends a command* to the machines that were assigned to it (`launch` action). When each machine receives it, it starts the command.
4. The available machines *acknowledge* the command (`ack2` action). The client counts the number of acknowledgement it receives.
5. The available machines *notify* the reservation system that the execution of its local process is done (`ack3` action). The client counts the number of acknowledgement it receives and, when it has received all of them, the execution is done.

4 Parameter synthesis and verification of the system

This system contains many parameters, and its behavior depends on these parameters. Parameter synthesis can be useful to verify is behavior under different parameters. An exploration of the behavior of such as system, for instance using behavioral cartography [7], can give the ranges of parameters for the system to behave as expected.

4.1 Expected behavior

The system must have several properties. The *soundness* of the system [5] (option to complete, proper completion and no dead transitions) was verified in [10, 11]: in the absence of failures, all the jobs are executed and complete. If resources can fail, there can be too many failures, in which case the jobs that need more resources than there are surviving resources cannot be executed. In this case, it was verified that *there exists* an execution in which all the jobs complete.

More specifically, it is also necessary to ensure *exclusive access* of the processes of all the jobs on the resources. It is one of the expected properties of this system: when a process of a job is executed on a resource, this resource must not be attributed to any other job. This property is important for instance for computation resources (computation nodes, GPUs...): if a node runs more processes than the number of cores it has, the processes need to share the execution time. This situation is called *oversubscription*. This property was also verified using Petri nets in [10, 11].

Timed models such as timed Petri nets [11] give a more precise idea of the behavior of the system in terms of, for example, deadlines. Verification techniques on such models can provide properties such as "in T time units, all the jobs have completed".

Parameter synthesis is particularly important to have a more precise idea of these properties. For instance, it can give precise completion time for a range of execution times. It is highly important for critical systems for example, to verify what is doable and under which conditions. In particular, one can want to make sure that for every possible execution, all the jobs complete before a certain number of time units. Parameter synthesis would help dimensioning the system (number of resources, number of jobs to put in the system) to be sure that the system behaves as required.

4.2 Parameters of the model

We have seen in sections 3.1 and 3.2 that the model depends on several parameters:

- The number of machines in the system;
- The number of clients issuing requests in the system;
- The number of resources asked by each client;
- The execution time of the processes of each job;
- The failure probability of each resource;
- In the *wait* semantics, the value of the timeout;
- In the *fail* semantics, the time after which the request is issued again.

The number of resources asked by each client is specific for each client. For instance, a client may ask for three machines and another one may ask for six machines. The execution time is assumed to be roughly the same for all the processes of a given job but not necessarily the same. As a consequence, machines spend some time in the state *finished*, but this time is generally small compared to the time spent in the state *running*.

However, if a resource has crashed during the execution of a process, the failed process must be re-executed, possibly from the beginning (some applications include a failure-recovery protocol that may reduce the re-execution time). Therefore, at the end of the execution of this job, the other resources used by the job are waiting for it in the state *finished*.

Therefore, the failure rate is very important to compute a likelihood of execution time. For instance, one can expect a result like "There is a likelihood of 50% that all the applications will be done after N time units, 25% after $2N$ time units, 15% after $3N$ time units and 10% that machines will crash too often for the applications to complete".

Besides, keeping some parameters unknown would allow to dimension the system with respect to some requirements. For instance, for a given number of resources, how many jobs can be executed and finish on time? Or the other way around, for a given number of jobs, how many resources does the system need to have to make sure that all the jobs will be executed and finish on time?

5 Conclusion

In this paper, we have presented a distributed algorithm for resource discovery and reservation. This algorithm was verified using model checking techniques (P/T Petri nets, Colored Petri nets and Timed Petri nets) in a previous paper, but these tools did not take into account the fact that the system contains several parameters, including on times and probabilities.

Parameter synthesis techniques such as behavioral cartography would permit to exhibit values for these parameters that would guarantee that the system follows a certain behavior. However, the complexity of the model, which is in the same time parametric, timed and probabilistic, makes it challenging for current parameter synthesis techniques.

Therefore, we hope that the system presented and modeled in this paper will be a challenge for the parameter synthesis community to find fitting parameter synthesis, maybe by hybridizing or combining several existing techniques, or developing new, specific ones.

One approach would be to consider parts of the problem as smaller instances and to decompose it as models of increasing complexity. A finite-state model, with no probability, can be used to perform a worst-case analysis. A real-time model, with parameterized delays, timeouts and number of processes

and resources but no probability, can be used to do a verification and quantification of the behavior of the system without failures. Last, the full model allows to perform verification and behavior analysis of the complete system.

Moreover, some assumptions can be made to perform a worst-case analysis and verify some properties. For instance, the failure probability can be simplified by a global failure rate, such as "at most 10% of the resources fail during the execution of a job". Under this assumption, the model loses its probabilistic nature.

References

[1] *LSF: Load Sharing Facility*. WWW Page: http://www.platform.com/Products/platform-lsf. Available at `http://www.platform.com/Products/platform-lsf`.

[2] *Portable Batch System*. WWW Page: http://www.pbsgridworks.com/. Available at `http://www.pbsgridworks.com/`.

[3] *Torque Resource Manager*. WWW Page: http://www.clusterresources.com/products/torque-resource-manager.php. Available at `http://www.clusterresources.com/products/torque-resource-manager.php`.

[4] *Zero Configuration Networking (Zeroconf)*. WWW Page: http://www.zeroconf.org. Available at `http://www.zeroconf.org`.

[5] W. M. P. van der Aalst (1997): *Verification of workflow nets*. In Pierre Azéma & Gianfranco Balbo, editors: *ICATPN'97*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 407–426, doi:10.1007/3-540-63139-9_48.

[6] Marcos Kawazoe Aguilera, Wei Chen & Sam Toueg (1997): *Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication*. In Marios Mavronicolas & Philippas Tsigas, editors: *Proceedings of the 11th Workshop on Distributed Algorithms (WDAG'97)*, *Lecture Notes in Computer Science* 1320, Springer, pp. 126–140, doi:10.1007/BFb0030680.

[7] Étienne André & Laurent Fribourg (2010): *Behavioral Cartography of Timed Automata*. In Antonín Kučera & Igor Potapov, editors: *Proceedings of the 4th Workshop on Reachability Problems in Computational Models (RP'10)*, *Lecture Notes in Computer Science* 6227, Springer, Brno, Czech Republic, pp. 76–90, doi:10.1007/978-3-642-15349-5. Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/AF-rp10.pdf`.

[8] Gerd Behrmann, Alexandre David & Kim G Larsen: *A tutorial on UPPAAL 4.0 (Updated November 28, 2006)*.

[9] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron & Olivier Richard (2005): *A batch scheduler with high level components*. In: *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGRID'05)*, IEEE Computer Society, Cardiff, UK, pp. 776–783, doi:10.1109/CCGRID.2005.1558641.

[10] Camille Coti, Sami Evangelista & Kais Klai (2015): *Queue-less, Uncentralized Resource Discovery: Formal Specification and Verification*. In Daniel Moldt, Heiko Rölke & Harald Störrle, editors: *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'15), including the International Workshop on Petri Nets for Adaptive Discrete Event Control Systems (ADECS 2015) A satellite event of the conferences: 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and 15th International Conference on Application of Concurrency to System Design ACSD 2015, Brussels, Belgium, June 22-23, 2015.*, *CEUR Workshop Proceedings* 1372, CEUR-WS.org, pp. 315–316. Available at `http://ceur-ws.org/Vol-1372/paper19.pdf`.

[11] Camille Coti, Sami Evangelista & Kais Klai (2015): *Time Petri Net Models for a New Queuless and Uncentralized Resource Discovery System*. *CoRR* abs/1502.03431. Available at `http://arxiv.org/abs/1502.03431`.