Design, Verification and Prototyping the Next Generation of Desktop Grid Middleware*

Leila Abidi[1,2], Christophe Cérin[1], and Kais Klai[1]

[1] Université de Paris 13, LIPN UMR CNRS 7030, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
[2] Université de Tunis, LaTICE ESSTT, 5 Avenue Taha Hussein, BP, 56, Bâb Manara, Tunis, Tunisie
{leila.abidi,christophe.cerin,kais.klai}@lipn.univ-paris13.fr

Abstract. This paper proposes a formal framework for the design and verification of a new Desktop Grid (DG) prototype which is currently developed with Web 2.0 technologies and only with this technology. The paper is an approach for developing a new generation of Desktop grid middleware, in our case based on Redis, a key-value no-SQL Web 2.0 tool with capability for managing the Publish-Subscribe asynchronous paradigm. We propose to revisit the Desktop Grid paradigm based only on concepts from Web 2.0 tools. It is different from previous approaches that have required to build software layers before the layer of the DG middleware. We demonstrate that this corresponds to a progress in freeing time for modeling and verification, that is, to build safe middleware. This work proposes (1) a modeling and a verification of a DG protocol based on the Publish-Subscribe paradigm (2) a prototype of a new generation of DG middleware that we are developing, concurrently with the modeling. A simulation, according to a prototype is conducted on a local cluster and demonstrate that our system is operational, light in terms of coding lines and used resources. Thus, it offers remarkable properties in order to implement DGs on tablets and Smartphones, we mean on resource constrained systems.

Keywords: Desktop grid computing, Grid middleware, Volunteer Computing, Service-oriented computing, resource management, Redis, Web 2.0, Publish-Subscribe paradigm, Formal Models, Colored Petri Nets.

1 Introduction

Desktop Grid [1] systems represent an alternative to supercomputers and parallel machines and they offer computing power at low cost. Desktop grids (DGs) are made with PCs and Internet as the communication layer. DGs aim to exploiting the resources of idle machines over Internet. Indeed, Desktop Grids have important features that explain the large number of international projects aiming to better exploit this computational potential. Many Desktop Grid systems

* Experiments presented in this paper were carried out using the Paris 13 experimental testbed.

R. Li, J. Cao, and J. Bourgeois (Eds.): GPC 2012, LNCS 7296, pp. 74–88, 2012.

have been developed using a centralized model. These infrastructures run in a dynamic environment and the number of resources may increase dynamically.

The Seti@Home [2] project is among the large amount of success stories. While the increasing number of users of such systems demonstrates the potential of Desktop Grid, current implementations, for instance Boinc [3], United Devices [4], Distributed.Net [5] and XtremWeb [6] still follow the client-server or master/slave paradigm. Theoretically, the computing power that can be obtained from these systems is constrained by the performance of the master node.

The BonjourGrid [7–9] middleware has appeared in this context. The basic idea is to exploit, dynamically, different instances of DG middleware. The coordination is fully distributed through Publish-Subscribe mechanisms. BonjourGrid is the first middleware of this kind, to the best of our knowledge, and it provides a view of the architecture and of the execution of applications running inside the middleware that match the ideas of decentralization.

In this paper, we provide graphic models based on Petri Nets to verify that the Publish-Subscribe system (BonjourGrid is just a case study introducing our Redis based prototype) is correct. Our research has been built around the desire to develop the BonjourGrid protocol using colored Petri Nets as a technique for modeling and verification and to receive a feedback about the good practices in developing DGs in the context of Web 2.0 tools.

This paper is organized as follows. After an introduction of the context of this work, we set in Section 2 our problem in the form of key issues that summarize the different aspects to what we are interested. Therefore, we introduce the principle of resource coordination (as implemented in BonjourGrid) and the benefit in using Publish-Subscribe systems. We also introduce our motivations for the design and the formal verification of BonjourGrid. We conclude this section by presenting some related work. Section 3 is related to our contributions, and describes the different steps of our work in order to provide a formal specification of the Publish-Subscribe paradigm. We also present our views to mix our Publish-Subscribe substrate with the core part of the BonjourGrid protocol, and finally we present the software prototype based on Redis. Section 5 concludes the paper.

2 Context and Motivations

2.1 Key Issues in Designing DG Middleware

In this section, we introduce the different issues facing the design of a DG middleware:

- Heterogeneity and volatility of resources are the main characteristics of DG environment that make communication, coordination, and scheduling difficult tasks. That's why we need a powerful mechanism in the middle of our system in order to guarantee a minimum of robustness and safety;
- The communication paradigm (for coordination, not for exchanging data) adopted should provide a high level of asynchronism in order to promote scalability; The question is: what is the appropriate model for controlling and coordinating the components of a DG middleware?

- The systems become so complex that we must think to verify them formally. This will allow us to have more confidence in what we code. We promote a co-design between specification and implementation parts, and we want to isolate pieces of code (the generic patterns) that will be generated automatically from the specification; The question is: how to specify and verify grid middleware?
- Web 2.0 technologies are the future, hence it would be a benefit to take advantage of them. On one hand, the Web 2.0 technologies assist to advertise desktop grid goals and attract computational resources for desktop grid communities. On the other hand, Web 2.0 systems should handle heavy data traffic and complex relations that need extraordinary large computational power: grid technologies. The question is: how Web 2.0 and grids technologies may merge?
- Grid technologies may serve as building blocks for Cloud technologies. In [10], we have explained how the DG paradigm is reused for the SlapOS system which is a provisioning and billing system for the cloud. SlapOS[1] is part of a 2.3M euros FUI project in which we are working on the coordination of servers. The question is to isolate problems in clouds that could be solved with grid technologies.

2.2 Resources Coordination

Desktop grids are characterized by a dynamic environment due to the heterogeneity and volatility of resources, in our case PCs at home. User's machines can join or leave the grid at any time, without any constraint. Each machine has its own properties such as its memory size, bandwidth, CPU/core number... that makes difficult the scheduling of tasks.

Consequently, the power of DGs that resides in the participation of volunteers, constitutes also a weakness in terms of resources orchestration when a job is submitted. Thus, the main problem with DGs is coordination, in particular when we have to execute communicating applications i.e., applications that are modeled by a task graph with precedence.

To bypass these problems, BonjourGrid counts on a distributed vision for the coordination and the execution of applications based on existing DG middleware. Moreover, the coordination mechanism is based on the Publish-Subscribe paradigm.

2.3 The Publish-Subscribe Paradigm

The Publish-Subscribe paradigm is an asynchronous mode for communicating between entities. Some users, namely the subscribers, or clients, or consumers, express and record their interests under the form of subscriptions, and are notified later by another event produced by other users, namely the producers [11].

[1] `http://www.slapos.org/`

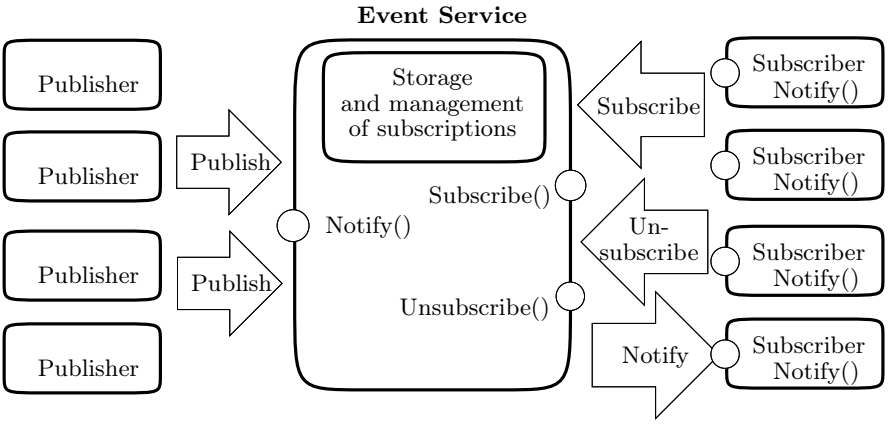

Fig. 1. The Publish-Subscribe paradigm [11]

As stated in Figure 1, subscribers record their interest by a call to the *subscribe()* operation inside the event service management system, without knowing the source of events. The *unsubscribe()* operation allows us to stop a subscription. The *notify()*(or *publish()*) operation is called by *publishers* in order to generate events that will be propagated to subscribers, and such events are managed by the event service management system too. Each subscriber will receive a notification for every event that is conform to its interest.

This communication mode is thus multi-point, anonymous and implicit. It is a multi-point mode (one-to-many or many-to-many) because events are sent to the set of clients that have declared an interest into the topic. It is an anonymous mode because the provider does not know the identity of clients. It is an implicit mode because the clients are determined by the subscriptions and not explicitly by the providers.

It is also known that this asynchronous communicating mode allows spatial decoupling (the interacting entities do not know each other), and time decoupling (the interacting entities do not need to participate at the same time). This total decoupling between the production and the consumption of services increases the scalability by eliminating many sorts of explicit dependencies between participating entities. Eliminating dependencies reduces the coordination needs and consequently the synchronizations between entities. These advantages make the communicating infrastructure well suited to the management of distributed systems and simplify the development of a middleware for the coordination of DGs.

2.4 BonjourGrid

BonjourGrid is an approach for the decentralization and the self organization of resources in DG systems [7–9]. The key idea is to exploit existing DG middleware (Boinc, Condor, XtremWeb) and concurrently to manage multiple instances of

DG middleware. The notion of meta desktop grid middleware has been introduced with BonjourGrid and the Publish-Subscribe paradigm is used intensively for the coordination of the different DG middleware.

Each user, behind a desktop machine in his office, can submit an application. BonjourGrid deploys a master (coordinator), locally on the user machine, and requests for participants (workers). Negotiations to select them should now take place. Using a Publish-Subscribe infrastructure, each machine publishes its state (idle, worker or master) when changes occur as well as information about its local load, or its use cost, in order to provide useful metrics for the selection of participants. Under these assumptions, the master node can select a subset of workers nodes according to a selection criteria. The master and the set of selected workers build the Computing Element (CE) that will execute and manage the user application. When the execution of an application of a CE terminates, its master becomes free, returns in the idle state, and it releases all workers who return to the idle state. Then, the nodes can participate to others projects.

To implement this approach, BonjourGrid has been decomposed in three fundamental parts: a) A fully decentralized resources discovery layer, based on Bonjour protocol [12]; b) A CE, using a Desktop Grid (DG) middleware such as XtremWeb, Condor or Boinc, which executes and manages the various tasks of applications; c) A fully decentralized protocol of coordination between a) and b) to manage and control all resources, services and CEs.

2.5 Related Work

Papers about Publish-Subscribe systems [13–15] are invitations to investigate more deeply the BonjourGrid protocol, in particular under the point of view of the verification of a distributed system. In this section we review the related works about publication-subscription systems and approaches for modeling and verifying formally such systems.

Publish-Subscribe systems concern both companies and researchers. Standards and industrial products are directly based on this paradigm, for instance the Bonjour protocol from Apple. For researchers, most of the works concern the problem of constructing a system as perfect as possible in terms of scalability, efficiency and safety. But they do not focus enough on the problem of the formal analysis of the accuracy of such systems. However, some research papers, such as [13–15], investigate the formal verification of Publish-Subscribe systems.

The work of Abidi and al. [16] focuses on the modeling of the BonjourGrid protocol. Authors have isolated the generic mechanisms of construction for the Publish-Subscribe approach. Then, they have modeled and verified, based on those mechanisms, the BonjourGrid protocol that allows the coordination of multiple instances of desktop grid middleware. Formal modeling allowed them to verify the adequacy of BonjourGrid with respect to the coordination of resources and to have a "composition" mechanism for integrating any protocol based on the Publish-Subscribe paradigm. All these ideas were illustrated along the BonjourGrid case study and they constitute a methodology for building Publish-Subscribe systems.

In this paper we continue this work, and we propose a more sophisticated formal model. We remind that in [16] we did not cover all the BonjourGrid protocol for the sake of simplicity whereas, in this paper we cover different internal details in order to have a realistic simulation of the actual behavior of the system.

In [13, 14], the authors propose an approach for modeling and validating systems. This approach is based on an architecture of components that react to events. In these works, the components are specified with UML state-transition diagrams. Formal verification is achieved through model checking (using SPIN). But instead of using the formulas of linear temporal logic (LTL) of SPIN, the authors have interpreted the properties as automata. According to them, this will represent more complex properties required to validate the modeled system.

Although our modeling approach is also based on component reacting on events, we do prefer to enjoy the advantages of temporal logic for the formal verification, in particular by using the ASK-CTL library.

In [15], the authors describe a generic framework dedicated to modeling and formal verification of Publish-Subscribe mechanisms. Their system is based on a model of states machine providing management of events during the execution of the publication-subscription protocol. The framework takes as input a set of components and a set of properties for the Publish-Subscribe mechanism. The matching of the two sets is subsequently validated using model checking tools. This system is regarded primarily as a generic framework in which there is always the risk of not providing a model and an audit tailored to each specific case for the Publish-Subscribe mechanism. Our approach is a successful modeling and formal verification. It is suitable for BonjourGrid while isolating the Publish-Subscribe mechanism.

In [17, 18], the authors, motivated by the benefits of formal analysis, build coordination protocol for a formal model using colored Petri Nets. To evaluate the accuracy of their model and as a result of their protocol, they checked the behavioral properties formally, and implemented a mechanism of CTL model checking. Our work is built around the protocol proposed by the authors. From our side, we also capitalize on the use of colored Petri Nets and CTL logic.

In [19], the authors present a new approach to modeling and formal verification, dedicated to software components. Their methodology is based on a software architecture-driven and the reuse of Petri Nets models. Their contribution is rather a new approach for visual composition, formal verification and validation of software systems. The work is built primarily around software components.

In [20], the authors present an overview of the analytical performance of colored Petri Nets in particular by using the CPN Tools. They use it to collect data during simulations, to generate different results on the performance and to implement several cases of simulation. A simple protocol is used to illustrate these aspects.

3 Contributions

3.1 Analysis and Criticisms

Modeling may guide the development of our forthcoming prototype that will serve for the validation of ideas and choices made during the design part.

The verification is for proving that the protocol is correct, then that any analyzed configuration will produce the "good" answer. This goal requires the formal verification of properties that we expect for the protocol: safety, for which the absence of deadlock is an example, and liveness.

In [16], we have modeled the BonjourGrid protocol according to the colored Petri Net [21] formalism and we have used the CPN Tools[2] for that purpose. CPN Tools is a fast and efficient simulator that handles both untimed and timed nets. Full and partial state spaces can be generated and analyzed, and a standard state space report contains information such as liveness properties. By means of a simple query language it is possible to specify and to check system specific properties.

The work introduced in [16] suffer from the fact that it is too specific to a dedicated protocol. In the current work, we have a more agnostic approach: we separate the Publish-Subscribe mechanisms and what is specific to the Bonjour-Grid protocol. The BonjourGrid serves as a guideline, a concrete example. We are looking for a "universal" Petri Net for the Publish-Subscribe paradigm on top of which any protocol based on Publish-Subscribe could be built and verified.

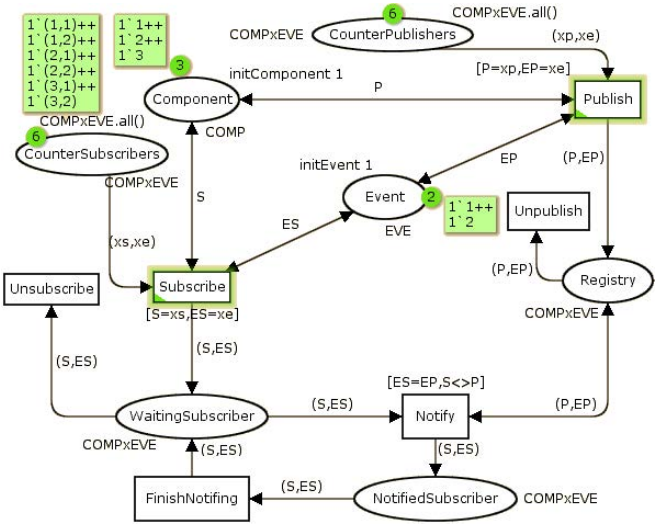

Fig. 2. Colored Petri Net for the Publish-Subscribe protocol

[2] See `http://www.daimi.au.dk/CPNtools`

3.2 A Colored Petri Net Model for the Publish-Subscribe Paradigm

For pedagogical reason, we start by explaining the Petri Net model we obtained for SCC in [16]. We show how the initial core idea was captured and it is essential to understand the concept and the model.

The colored Petri Net model in Figure 2 models the Publish-Subscribe paradigm. It introduces an "initial" state with a definite number of components and events. Each component can be a publisher (represented by **EP** for EventPublished on the figure), or a subscriber (represented by **ES** for EventSubscribed on the figure) or both.

This model is compliant with the Publish-Subscribe protocol, since a component can publish an event as many times as it wants; it can also subscribe to an event as many times as it wants. An event can be issued by one or more components, and one or more components can subscribe to this event.

Published events are saved in a directory that is modeled by the place "Registry". When a component subscribes to an event **E**, it goes to the place "WaitingSubscriber". Once the event is published, the transition "Notify" can be fired. A condition should be checked when we fire the transition "Notify": "a component cannot subscribe to the event it published", which was modeled using the guard $[S <> P]$.

Hence, we have successfully achieved our first aim by defining a colored Petri Net for which we can build representations of any protocol that is written with the publish-subscribe paradigm in mind. In the next section, we model the BonjourGrid protocol built on top of that Publish-Subscribe Petri Net model.

3.3 A Colored Petri Net Model for the BonjourGrid Protocol

Figure 3 illustrates the current methodology used to compose any Publish-Subscribe protocol. This figure is related to the central part of the BonjourGrid protocol and constitutes a contribution of this paper. Indeed, this paper exhibits a major refinement of the initial specification as published in [16].

The new refinements cover different internal details of the BonjourGrid protocol around the central part. The difficulty for the designer is to have a global view of the overall behavior of such a system where asynchronism is the essential criterion.

In Publish-Subscribe architectures, components communicate with each other through the exchange of events. Thus, any model of these architectures must explicitly consider the two main actors: components and events, and the three main services: publish, subscribe and notify.

The first actor stands for publishing an event **ep** from a machine **c**, the second one is for subscribing an event **es** coming from a machine **c**, and the third one is for notifying events to interested machines.

BonjourGrid modeling is focused on that aspect.

In fact, the challenge was to consider a "black box" (the Petri Net for the Publish-Subscribe model which is represented in the center of Figure 3) that cannot be modified and to try to specify the behavior of BonjourGrid as external events of the black box. By doing this, we exhibit a general methodology.

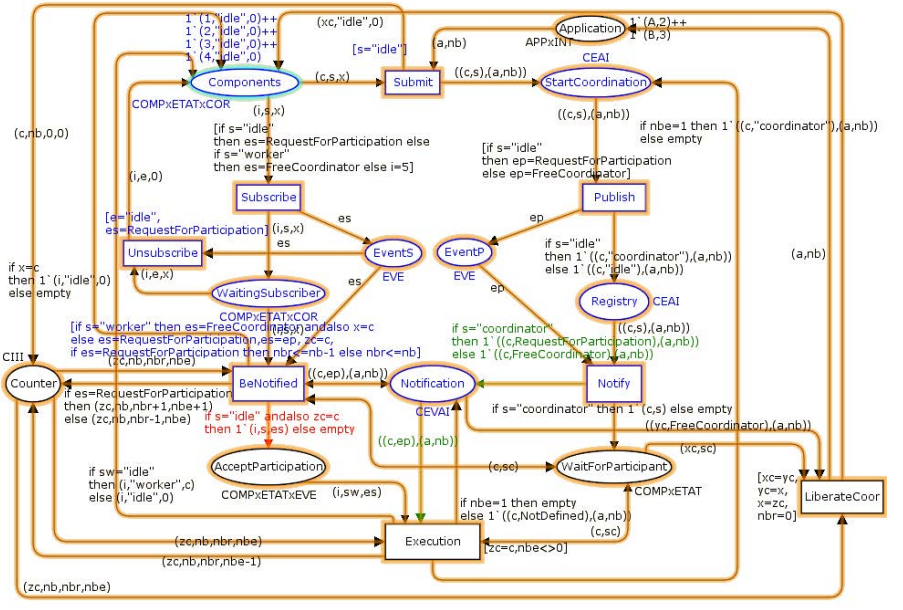

Fig. 3. BonjourGrid model, composed on top of a Publish-Subscribe Petri Net

The main idea is to plug the BonjourGrid protocol on top of the Publish-Subscribe protocol. The later is mainly presented by the cycle: "publication", "subscription" and "notification". BonjourGrid elements are plugged on inputs and outputs of this cycle.

In BonjourGrid model we just blow up the state "Event", previously represented in Publish-Subscribe model, into two states "EventS" and "EventP" to differentiate between events that are published and those for which components can subscribe, in order to have more clarity.

We represent each component in the place *component* by the tuple (compId, state, coorId) where compId is the identifier of the component, state can take the values : "idle", "worker", "coordinator" that represent the different states a component may take, coorId is the identifier of the coordinator attached to the component/worker; it takes the value 0 when the component is not a worker.

Firstly, a component may submit an application **a**, and publish a "RequestForParticipation" event associated to that application. the state *Application* is represented by the couple **(a,nb)** where **a** is the identifier of the application and **nb** is number of participant required to execute this application.

In parallel, other components (in "idle" states) may subscribe to that "RequestForParticipation" event. A coordination starts when subscribed machines are notified by their corresponding events and accept to participate to the execution of the application. All the subscribers on that event are notified by its publication, but only the number **nb** required by the application can move to

"AcceptParticipation" state to assist in its execution. The place "counter" serves to ensure that task. In this step, the state "idle" becomes "worker". Coordinators who have required in their support applications a number of participants that is not provided yet, must wait in the place "WaitForParticipant" until the required number **nb** becomes available (simulated with the place "WaitForParticipant"). The execution of the application can start when the required number **nb** of participants is reached. All these coordinating steps will be locked in the place "WaitForParticipant" until the application is completed.

Once the execution is completed, we move to the step of releasing the coordinator and the components that are attached. Thus, workers may subscribe to the "FreeCoordinator" event that would allow to release them. When coordination is finished, the coordinator component publishes the "FreeCoordinator" event then subscribed workers are notified. They are released and the state moves from "worker" to "idle".

By doing this, we have successfully achieved our second aim by formally modeling the BonjourGrid protocol built on top of that Publish-Subscribe Petri Net. Using CPN Tools, we then performed simulations of the net to gain more confidence in our model. During this first step of the simulation, no straightforward problems was discovered. The next step was naturally to perform an exhaustive simulation exploring all the possible states of the system, i.e., its state space.

For the verification of the desired properties (liveness for instance), our goal was to keep the same level of abstraction as the modeling of the CPN. For these reasons, we did not use any temporal logic formalisms since they work at a different level of abstraction and, thus, it can be difficult to use. We exploited the tools provided by CPN Tools, which allow us to calculate and analyze state spaces. With these tools, the standard queries for the verification require no programming at all.

Fundamental properties we want to verify on that model were:

- Any event produced (published) must be received by all interested consumers (subscribers).
- A coordinator **C** begins execution of its implementation if and only if there exists at least one machine **M** that agree to participate with **C**.
- If a coordinator **C** publishes the event "FreeCoordinator" then all Workers for that coordinator will eventually switch to the "idle" state.
- Any worker **W** can be attached to a single coordinator **C**.

Figure 4 provides a sample of the state space report provided by CPN Tools.

The analysis of these small configurations gives us more confidence in the BonjourGrid system especially considering the following facts:

- We have not found any deadlock states (i.e., states that do not admit executable transitions). The absence of such state is obviously required in our context.
- All possible transitions are executable. Hence, our specification seems to be correct from the perspective of event triggering: all possible events can eventually happen.

– All state spaces built are composed of a single strongly connected component. It seems therefore impossible to be trapped in a specific or undesired system configuration. This liveness property is indeed a crucial prerequisite for our system.

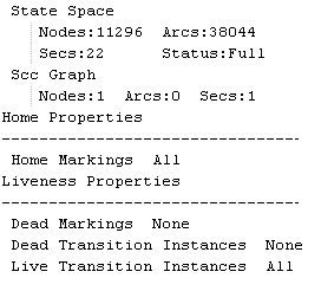

Fig. 4. A partial view of the report of the BonjourGrid model

3.4 A Prototype Based on Redis

In parallel with the modeling and verification parts, as introduced in the previous section, we have built a prototype of a Desktop Grid middleware based on BonjourGrid and Redis and on top of Python. Redis[3] is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets. Moreover, it implements a Publish-Subscribe layer. Indeed, we have an "all-in-one" tool that fulfills our initial needs: storage of codes we have to execute, storage of input/output data, support for Publish-Subscribe. This explains why we have declined the opportunities to work with XMPP, Nodejs like tools, or the promising Hookbox interface which is a Comet server and message queue that tightly integrates with Web application frameworks.

Redis do not offer a strong support for authentication, cryptographic communication or data protection but it is well targeted for our work related to prototyping. We assume that in the future, the community developers will offer such properties. However, notice that Redis supports some sort of server redundancy to make the system fault tolerant.

Our prototype does not include yet all the components of DG middleware such as those we find in Boinc, Condor, or XtremWeb. For instance we do not have a component for doing result certification or we do not manage fault tolerance issues by task duplication. Again, the prototype is currently devoted to the understanding of the core decentralized protocol for the coordination of resources and it would be too challenging to hope that we will solve all the problems in one shot. We do prefer to work “step by step”.

Our prototype is organized according to the following Python classes:

[3] See `http://www.redis.io`

- ServerClass. There are three possibilities for servers: the main server for the protocol itself, the data server storing input/output data for the applications and CodeServer which is the name of the server for retrieving codes of applications; The three servers can point on the same name;
- DataManager: set-up an instance of ServerClass; In this class, we also define functions for loading files into a Redis server, for executing a code (binary or script);
- FormMultiprocessingTasks: it is the core of the "workflow" engine because the method in this class "executes" the task graph; Note that we create threads to execute, concurrently, all the tasks (descendants in the task graph) attached to a node;
- EngineClass. It starts and instance of the FormMultiprocessingTasks class in one thread, and wait that all nodes of the task graph are visited in another thread;
- MachineClass: it allows to set the properties of a machine (operating system type, amount of memory on the machine, processor type...); In this class we have also two methods for launching a worker and a coordinator respectively;
- WorkerClass and CoordinatorClass define the behavior of a worker, or a coordinator;

Our prototype executes series-parallel graphs (SPG). Intuitively, a SPG is built from a sequence of compositions (parallel or series) of smaller-size SPGs. The smallest SPG consists of two nodes connected by an edge. The first node is the source of the SPG while the second is its sink. When composing two SPGs in series, we merge the sink of the first SPG with the source of the second. For a parallel composition, the two sources are merged, as well as the two sinks. In our prototype we execute the graph depicted on Figure 5.

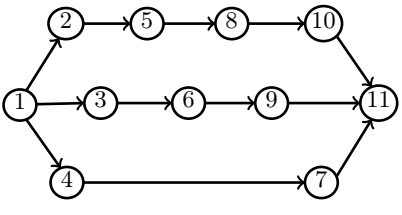

Fig. 5. The graph executed by our prototype. The initial state is 1, the final state is 11.

Classical workflow applications usually consists of a directed acyclic graph: the application is made of several tasks, and there are dependencies between these tasks. However, it turns out that many of these graphs are series-parallel graphs. For instance, in [22], McClatchey et al. introduce a prototype scientific workflow management system called CRISTAL, and the distributed scientific workflow applications that they consider are SPGs. In [23], Qin and Fahringer

use scientific grid workflow applications, which are all structured as SPGs: the WIEN2k workflow performs electronic structure computations of solids; the MeteoAG workflow is a meteorology simulation application [24], and the GRASIL workflow computes the spectral energy distribution of galaxies [25]. A last, the fMRI workflow [26], which is a cognitive neuroscience application is also based on SPGs.

The simulation implemented in the prototype[4] is as follows. A series-parallel graph with 11 vertices is defined. Each edge represents an application. The application is the same for all edges: it is a Bash script that should be located on /tmp and echoing a message.

The program forks and we start a coordinator as well as a worker (we use only one worker which is created for executing one application, one by one). The program is multi-threaded in the sense that we execute the descendants of a node (the applications) in different threads. We manage locks so that we cannot predict which is the order of execution of the descendants.

The coordinator is in a loop doing the work described in this paragraph. It publishes a message to advertise that he needs a worker for executing a task (according to some properties). Worker(s) listen on the channel dedicated to this message, then reply by publishing a random channel name for future communication between him and the coordinator. The coordinator accepts the first response (the worker who is arriving the first) and he publishes on the random channel name the name and the location of the application. The worker keeps the application name and executes it.

It is important to notice that the protocol is entirely depicted by the exchange of publication and subscribe messages and the development is guided by our different modeling. The prototype serves for the validation of ideas and choices made during the design and modeling parts.

Note also that in the prototype we do not manage files representing the input/output data of the applications. This point is related to scheduling strategies and we plan to include a more elaborated scheduling class in our prototype in the future. In the context of Desktop Grid, scheduling should serve also to check the results computed on (hostile) workers. The issue is to write scheduling algorithms in terms of the Publish-Subscribe paradigm and in such a way that the strategies could be composed with the current Petri Net.

4 Conclusion

In this paper, we have introduced the context of our work about the coordination of resources using the Publish-Subscribe paradigm. We have also demonstrated the usefulness of modeling and formal verification of such a specific mechanism for the BonjourGrid system, dedicated to the management of multiple instances of Desktop Grid middleware.

This work is a step toward the development of DG middleware based on Web 2.0 technologies. Furthermore, this effort has been consolidated in this paper

[4] See `http://www.lipn.fr/~cerin/ProtoRedis.tar`

with another facet of the problem, namely the definition of a mechanism for composition of the basic scheme introduced in this paper with any protocol that is written with the Publish-Subscribe paradigm in mind. Recall that BonjourGrid is only a case study that allows us to discuss the problems and devise solutions that we have implemented, concurrently with the design, in Redis.

We are currently working on the programming effort for introducing a Python module for scheduling jobs and later, we will conduct experiments on large clusters running Redis. Scheduling is an important issue because we want to do result certification: the computation are done on (hostile) workers so we need to do redundant computation. We have imagined an algorithm based on tickets that we duplicate and managed by the Publish-Subscribe paradigm. Again, the scheduling itself is made exclusively through a Publish-Subscribe approach which is unconventional, but allows to build a fully distributed protocol where asynchronism is maximized because of scalability requirement of systems we build nowadays.

References

1. Kondo, D.: Preface to the special issue on volunteer computing and desktop grids. J. Grid Comput. 7, 417–418 (2009)
2. University of California: SETI@Home (October 2011), `http://setiathome.berkeley.edu/`
3. University of California: BOINC (October 2011), `http://boinc.berkeley.edu/`
4. Univa: United Devices (October 2011), `http://www.unicluster.org/`
5. DistributedNet: Distributed.Net (October 2011), `http://www.distributed.net/`
6. Univa: XtremWeb (October 2011), `http://www.xtremweb.net/`
7. Abbes, H., Cérin, C., Jemni, M.: Bonjourgrid as a decentralised job scheduler. In: APSCC, pp. 89–94. IEEE (2008)
8. Abbes, H., Cérin, C., Jemni, M.: Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids. In: IPDPS, pp. 1–8. IEEE (2009)
9. Abbes, H., Cérin, C., Jemni, M.: A decentralized and fault-tolerant desktop grid system for distributed applications. Concurrency and Computation: Practice and Experience 22, 261–277 (2010)
10. Smets-Solanes, J.P., Cérin, C., Courteaud, R.: Slapos: A multi-purpose distributed cloud operating system based on an erp billing model. [27] , 765–766
11. Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35, 114–131 (2003)
12. Cheshire, S., Steinberg, D.H.: Zero configuration networking - the definitive guide: things that just work: covers Apple's Bonjour APIs. O'Reilly (2005)
13. Zanolin, L., Ghezzi, C., Baresi, L.: An approach to model and validate publish/subscribe architectures (2003)
14. Harrison, M.D., Kray, C., Sun, Z., Zhang, H.: Factoring user Experience into the Design of Ambient and Mobile Systems. In: Gulliksen, J., Harning, M.B., van der Veer, G.C., Wesson, J. (eds.) EIS 2007. LNCS, vol. 4940, pp. 243–259. Springer, Heidelberg (2008)
15. Garlan, D., Khersonsky, S., Kim, I.: Model Checking Publish-Subscribe Systems. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 166–180. Springer, Heidelberg (2003)

16. Abidi, L., Cérin, C., Evangelista, S.: A petri-net model for the publish-subscribe paradigm and its application for the verification of the bonjourgrid middleware. [27], 496–503
17. Kacem, N.H., Kacem, A.H., Jmaiel, M., Drira, K.: Towards modelling and analysis of a coordination protocol for dynamic software adaptation. In: Chbeir, R., Badr, Y., Abraham, A., Laurent, D., Köppen, M., Ferri, F., Zadeh, L.A., Ohsawa, Y. (eds.) CSTST, pp. 499–507. ACM (2008)
18. Kacem, N.H., Kacem, A.H., Drira, K.: A formal model of a multi-step coordination protocol for self-adaptive software using coloured petri nets. International Journal of Computing and Information Sciences (2009)
19. Silva, L.D.D., Perkusich, A.: Formal verification of component-based software systems. In: Isaías, P.T., Sedes, F., Augusto, J.C., Ultes-Nitsche, U. (eds.) NDDL/VVEIS, pp. 113–124. ICEIS Press (2003)
20. Wells, L.: Performance analysis using cpn tools. In: Lenzini, L., Cruz, R.L. (eds.) VALUETOOLS. ACM International Conference Proceeding Series, vol. 180, p. 59. ACM (2006)
21. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems, 1st edn., July 1. Springer, Heidelberg (2009)
22. McClatchey, R., Estrella, F., Le Goff, J.M., Kovacs, Z., Baker, N.: Object databases in a distributed scientific workflow application. In: Proceedings of the 3rd Basque International Workshop on Information Technology (BIWIT 1997), p. 11. IEEE Computer Society, Washington, DC (1997)
23. Qin, J., Fahringer, T.: Advanced data flow support for scientific grid workflow applications. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 42:1–42:12. ACM, New York (2007)
24. Schüller, F., Qin, J., Nadeem, F., Prodan, R., Fahringer, T., Mayr, G.: Performance, scalability and quality of the meteorological grid workflow meteoag. In: Proceedings of the 2nd Austrian Grid Symp., Univ. Innsbruck (2006)
25. Silva, L., Granato, G.L., Bressan, A., Lacey, C.G., Baugh, C.M., Cole, S., Frenk, C.S.: Modeling dust on galactic sed: Application to semi-analytical galaxy formation models (1999)
26. Zhao, Y., Wilde, M., Foster, I., Voeckler, J., Jordan, T., Quigg, E., Dobson, J.: Grid middleware services for virtual data discovery, composition, and integration. In: 2nd Workshop on Middleware for Grid Computing, p. 57. ACM Press (2004)
27. Jacobsen, H.A., Wang, Y., Hung, P. (eds.): IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, July 4-9. IEEE (2011)