Decentralized Enactment of BPEL Processes

Michael Pantazoglou, Ioannis Pogkas, and Aphrodite Tsalgatidou

Abstract—This article presents BPELcube, a framework comprising a scalable architecture and a set of distributed algorithms, which support the decentralized enactment of BPEL processes. In many application domains, BPEL processes are long-running, they involve the exchange of voluminous data with external Web services, and are concurrently accessed by large numbers of users. In such context, centralized BPEL process execution engines pose considerable limitations in terms of scalability and performance. To overcome such problems, a scalable hypercube peer-to-peer topology is employed by BPELcube in order to organize an arbitrary number of nodes, which can then collaborate in the decentralized execution and monitoring of BPEL processes. Contrary to traditional clustering approaches, each node does not fully take charge of executing the whole process; rather, it contributes to the overall process execution by running a subset of the process activities, and maintaining a subset of the process variables. Hence, the hypercube-based infrastructure acts as a single execution engine, where workload is evenly distributed among the participating nodes in a fine-grained manner. An experimental evaluation of BPELcube and a comparison with centralized and clustered BPEL engine architectures demonstrates that the decentralized approach yields improved process execution times and throughput.

Index Terms—Composite Web Services, Processes, Business Process Management, Simulation of Business Processes

1 Introduction

The *Web Services Business Process Execution Language* [1], abbreviated to WS-BPEL or BPEL, is widely considered the *de facto* standard for the implementation of executable service-oriented business processes as compositions of Web services. The language specification defines a set of *activities* to support synchronous and asynchronous interactions between a process and its clients, as well as between a process and external Web services. Moreover, a number of *structured* activities are used to implement typical control flow units such as sequential or parallel execution, if-else statements, loops, etc. Hence the control flow of a business process is realized by a number of activities, which are appropriately ordered and put together. The BPEL language also provides the necessary elements to support the expression of common programming concepts such as scope encapsulation, fault handling, compensation, and thread synchronization.

Data handling is realized by means of *variables*, which are conveniently used by a BPEL process to hold the data that are generated and/or consumed upon execution of its constituent activities. Thus the various activities of a process are able to share data with each other simply by reading from and writing to one or more of the process variables.

To date, most of the available solutions for the execution of BPEL processes have been designed and operate in a centralized manner, whereby an orchestrator component running on a single server is responsible for the execution of all process instances, while all relevant data are maintained at a single location (i.e. the server hosting the BPEL engine). Clearly, such approach cannot scale in the presence of a potentially large number of simultaneous, long-running process instances that produce and consume voluminous data. While in some cases clustering techniques are supported and can be employed to address the scalability issue, the deployment and maintenance of clusters consisting of two or more centralized BPEL engines sets requirements on the underlying hardware resources, which cannot be always fulfilled by the involved organizations. Furthermore, clustering could be proved an inefficient approach under certain conditions, as it cannot overcome the emergence of bottlenecks that are caused by specific activities of a BPEL process. Hence, a more fine-grained workload distribution approach is called for to ensure scalability of the BPEL execution engine at lower cost.

In the following section, we introduce a motivating scenario from the environmental domain so as to better capture the problem in real-world terms.

1.1 Motivation

In order to facilitate the development, delivery and reuse of environmental software models, service orientation has been recently pushed forward by several important initiatives[1,2,3] and international standardization bodies[4] in the environmental domain. In the light of those efforts, both geospatial data and geo-processing units are exposed as Web services, which can be used as building blocks for the composition of environmental models in the form of BPEL processes [2], [3], [4], [5]. However,

• *The authors are with the Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilissia, Athens 15784, Greece.*
E-mail: {michaelp,jpogkas,atsalga}@di.uoa.gr

1. INSPIRE, http://inspire.jrc.ec.europa.eu/
2. GMES, http://www.gmes.info
3. SEIS, http://ec.europa.eu/environment/seis/
4. OGC, http://www.opengeospatial.org/

several challenges arise upon this paradigm shift. Efficient execution and monitoring of long-running environmental processes that consume and produce large volumes of data, in the presence of multiple concurrent process instances are among the prominent issues that one should effectively deal with.

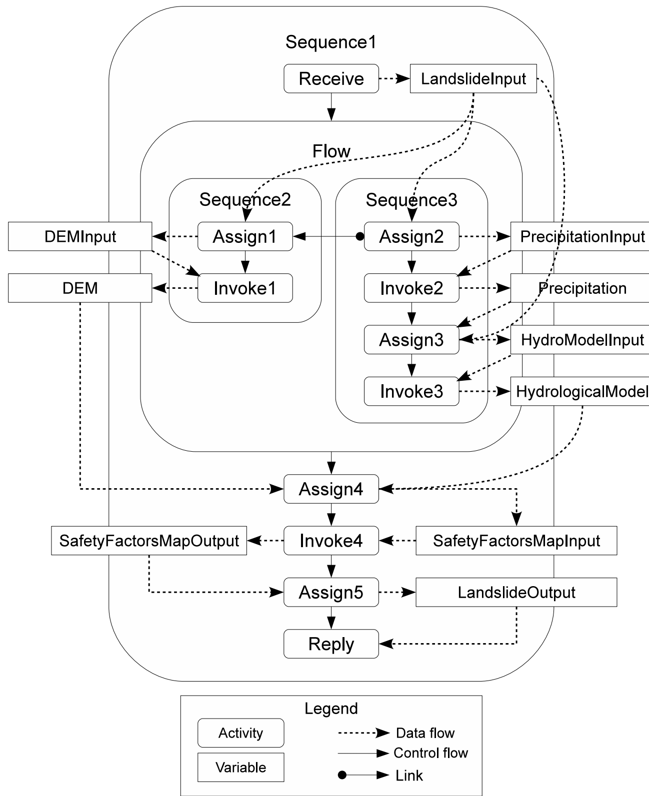

Fig. 1. A BPEL process used for the calculation of landslide probabilities in a user-specified area.

Let us consider the BPEL process of Figure 1, which was developed by one of our industrial partners in the ENVISION project [6], and is part of a decision support system dedicated to landslide risk assessment. Overall, the process consists of 15 activities, which are represented in the diagram as rounded boxes, and 10 variables, which are shown as cornered boxes. The control flow of the process is indicated by normal arrows connecting the various activities, while the dashed arrows pointing from the variables to the activities and vice versa display its data flow. The various *assign* activities in the process are used to copy data from one variable to another; the *invoke* activities allow the process to interact with external Web services; the *receive* and *reply* activities are used by the process in order to retrieve the user input and send the final output, respectively; finally, the structured *sequence* and *flow* activities dictate the order in which their included activities will be executed.

In essence, the landslide process orchestrates four Web services. First, a digital elevation model (DEM) of the user-specified area is retrieved by invoking an appropriate service through activity *Invoke1*. In parallel, a sensor service is called through activity *Invoke2* in order to retrieve the precipitation data of the user-specified area. These data along with a set of user input parameters are further fed through activity *Invoke3* to a processing service, which simulates the main mechanisms of the water cycle by a system of reservoir. The produced digital elevation model and the hydrological model containing the produced map of groundwater level in that area are finally passed as input to another processing service, which is invoked through activity *Invoke4* and performs static mechanical analysis, in order to calculate the landslide probabilities in the area of study, in the form of a map of safety factors ranging between zero and one. That map is finally returned to the user as the process response.

According to our partner, the execution time of the BPEL process implementing the landslide model may range from a few seconds/minutes to many days, depending on the addressed area and the desired level of detail. Those delays are usually introduced by the external Web services, and force the process instances to remain alive consuming valuable resources of the BPEL engine. Such long-running instances also involve the generation, consumption and processing of large data sets ranging between a few kilobytes and hundreds of megabytes, such as the sensor observations, the digital elevation model, etc. Moreover, due to their potentially long duration, clients need to monitor the execution progress by retrieving intermediate results (e.g. the external Web service outputs) from the execution engine. As the number of clients increases, the resources of the centralized infrastructure hosting the executable process are quickly exhausted, and the BPEL engine becomes bloated with requests coming from multiple concurrent users. Hence, the overall throughput of the execution infrastructure is dramatically deteriorated, while the process execution times escalate at unacceptable levels.

Looking beyond the above scenario, one may find that, similar scalability and performance issues have been observed in many other application domains where BPEL is used, such as supply chain management, online retail, or healthcare [7]. BPEL applications related to synchronous multiple bookings, or asynchronous composite searching and downloading [8] have also shown the limitations of centralized BPEL engines. Going back to our motivating example, all the aforementioned conditions currently prevent the exposure and reuse of the landslide model on the Web. Our partner cannot afford expensive servers to host a cluster of BPEL engines, while the use of Cloud computing solutions [9] is prohibited due to administrative and security policies. Instead, an ideal solution to the identified scalability and performance problems would be to distribute the execution of the process activities and the maintenance of the process variables across the existing infrastructure, which consists of less expensive equipment with limited hardware resources.

1.2 Contribution

In an effort to address situations such as the one described previously, we introduce a framework comprising a scalable Peer-to-Peer (P2P) architecture and a set of distributed algorithms to support the decentralized enactment of BPEL processes. Our framework, dubbed BPELcube hereinafter, particularly focuses on the improvement of the average process execution times, and the enhancement of the overall throughput of the execution infrastructure in the presence of multiple, concurrent and long-running process instances. BPELcube is mainly characterized by the following features:

- *Fully decentralized, P2P-based BPEL engine architecture*. BPEL processes are deployed, executed, and monitored by a set of nodes organized in a hypercube P2P topology. Each node does not fully take charge of executing the whole process; rather, it contributes by running a sub-set of the process activities, and maintaining a sub-set of the generated process data. Thus the BPEL execution engine is fully operational without the need of any central controller components.
- *Fine-grained distribution of process activities*. Decentralization of process execution fits to the nature of long-running business-to-business interactions, and significantly improves the performance and throughput of the execution infrastructure. BPEL processes are fully decomposed into their constituent activities. Large-scale parallelization is feasible as the various activities designated to run in parallel can be synchronized and executed by different nodes.
- *Proximity-based distribution of process variables*. Since in many application domains processes consume and produce large volumes of data, it is important that those data are distributed in order to avoid resource exhaustion situations. Our algorithms make sure that the data produced by a BPEL process will be distributed across the nodes involved in its execution. Moreover, they will stay close to the process activities that produce them, thereby avoiding the unnecessary transfer of potentially large volumes of data between nodes as much as possible.
- *Asynchronous interaction with the client*. Even if a BPEL process is synchronous following the request-response communication pattern, the interaction between the client and the distributed execution engine occurs in an asynchronous, non-blocking manner. This way, the execution engine is able to serve multiple long-running process instances without the need to maintain open connections to the respective clients over long periods of time. Furthermore, while waiting for a long-running process instance to complete, clients are given the monitoring mechanisms to retrieve intermediate results, without intervening or inflicting additional delays to the process execution.
- *Efficient use of the available resources and balanced workload distribution*. The BPELcube algorithms ensure that all nodes available in the P2P infrastructure will contribute to the execution of BPEL processes. The frequency of use of each node is taken into account upon load balancing, while efficient routing techniques are employed in order to achieve an even distribution of the workload at any given time and thereby avoid the emergence of performance bottlenecks.

In the following section, we present an analysis of the relative literature and pinpoint the added value of our work in the context of decentralized BPEL process execution. Then, we proceed in Section 3 with the detailed presentation of our proposed approach. Examples based on the landslide BPEL process that was described in Section 1.1 are given where necessary in order to better explain the various algorithms. An experimental evaluation of our approach along with the retrieved measurements are presented and discussed in Section 4, while we conclude this paper and identify paths for future work in Section 5.

2 Related Work

2.1 Distributed Scientific Workflow Systems

In the last years, developments in Scientific Workflow (SWF) systems have made possible the efficient, scalable execution of long-running workflows that comprise large numbers of parallel tasks, and operate on large sets of data. Since the challenges met by those efforts bear some resemblance to the motivation behind BPELcube, we would like to emphasize on their different scope and technical foundations.

By definition, the majority of SWF solutions are particularly designed to support the modeling and execution of *in silico* simulations and scientific experiments. Moreover, they are mostly based on proprietary executable languages, which are tailor-made to the needs of such applications. On the other hand, by using the BPEL standard as its underlying basis, BPELcube has more general purposes and can be used to support a wider range of environments and applications. The different scopes of BPELcube and the various SWF systems are also reflected by their pursued programming models. Due to their data-flow orientation, most SWF engines (see for instance Taverna [10]) follow a functional, data-driven model, whereas BPEL engines, including BPELcube, implement an imperative, control-driven model. Hence, the focus of BPELcube is on implementing algorithms that distribute the control flow of processes, in a way that no central coordinator is required. On the other hand, since the control flow is of minor importance to scientific workflows, SWF systems build on efficient parallelization and pipelining techniques, in order to improve the processing of large-scale data flows. For instance, Pegasus [11] attains scalability by mainly addressing the large number of parallel tasks in a single workflow,

and the corresponding voluminous data sets, through task clustering and in-advance resource provisioning. In BPELcube, we are primarily interested in improving the throughput of the BPEL engine, and the average process execution times, in the presence of large numbers of concurrently running process instances that are long-running, and consume potentially large data sets.

Most SWF systems (e.g. Kepler [12], Triana [13], or Pegasus [14]) exhibit a clear separation of concerns between the design of a workflow and the execution infrastructure, although much effort has been spent on supporting Grid settings such as Globus. In general, however, Grid infrastructures are heavy-weight, complex, and thus difficult to manage and maintain. In contrast, BPELcube is able to seamlessly organize and manage any set of nodes in a hypercube topology, so as to engage them in the execution of long-running and resource-demanding processes.

Still, despite their inherently different scopes, programming models, and scalability concerns, SWF systems have effectively dealt with advanced data management aspects, such as provenance [15], or high-speed data transfer [16]. These features are complementary to our approach and could be accommodated by BPELcube to further enhance its capabilities and performance.

2.2 BPEL Decentralization

The decomposition and decentralized enactment of BPEL processes is a valid problem that has been the subject of many research efforts in the last years. In the following, we review a number of related results that have become available in the literature.

A P2P-based workflow management system called SwinDeW that enables the decentralized execution of workflows was proposed by Yan et al. [17]. According to the authors, the generic workflow representation model is compatible with most concrete workflow languages including BPEL, although this compatibility is not demonstrated. In any case, similar to our presented approach, SwinDeW is based on the decomposition of a given workflow into its constituent tasks, and their subsequent assignment to the available nodes of a P2P network, in order to remove the performance bottleneck of centralized engines. Unlike BPELcube, however, there is no evaluation of SwinDeW to assess its applicability in settings where processes are long-running and produce large volumes of data. Besides, a main difference between the two approaches lies in their corresponding worker recruitment algorithms: SwinDeW makes use of the JXTA [18] peer discovery mechanism to find nodes with specific capabilities, and then quantifies their workload before assigning the given task to the one with the minimum workload. Since the respective discovery protocol cannot guarantee that all relevant peers will be found upon a query, it may become possible that not all available nodes in the P2P network are equally utilized. In BPELcube, the recruitment algorithm relies on the hypercube topology, the inherent ability to perform efficient random walks, and the frequency of use of each node in order to evenly divide the workload and thereby exploit all available resources.

The NIÑOS orchestration architecture [7] is based on a distributed content-based publish/subscribe (pub-/sub hereinafter) infrastructure, which is leveraged to transform BPEL processes into fine-grained pub/sub agents. The latter then interact using pub/sub messages and collaborate in order to execute a process in a distributed manner. A critical departure of BPELcube from the NIÑOS approach lies in the respective process deployment mechanisms. In NIÑOS, a BPEL process is deployed prior to its execution to a number of agents, which remain the same for all subsequent executions of the process. Hence, the infrastructure may underperform in the presence of multiple concurrent instances of the same process. In our case, the BPEL process is decomposed and its constituent activities are assigned to the available nodes in the P2P network at runtime, depending on their current workload which is inferred by their frequence of use. Furthermore, the evaluation of NIÑOS has been performed in a more relaxed setting in terms of message sizes and external service delays, which does not give any evidence of the applicability of that approach to long-running processes producing big data.

In an attempt to improve the throughput of the BPEL process execution infrastructure in the presence of multiple concurrent clients, a program partitioning technique has been proposed by Nanda et al. [19], which splits a given BPEL process specification into an equivalent set of processes. The latter are then executed by different server nodes without the need of a centralized coordinator. Similar approaches have also been proposed by Baresi et al. [20] as well as by Yildiz and Godart [21]. Along the same lines, the use of a penalty-based genetic algorithm to partition a given BPEL process and thereby allow decentralized execution was proposed by Ai et al. [22]. However, to realize these partitioning techniques, each participating node must host a full-fledged BPEL engine, which is often heavyweight and therefore not always affordable by many small organizations and businesses. In our approach, there is no such requirement imposed on the nodes forming the underlying P2P infrastructure, and thus each node has a relatively small memory footprint. This way, our distributed BPEL engine can leverage and be deployed on hardware with limited capabilities, as it is already demonstrated by the experimental setting described in Section 4.

A solution to the problem of decentralized BPEL workflow enactment that is based on the use of tuplespace technology was reported by Wutke et al. [23]. According to that approach, workflows are defined as BPEL processes, which are split among all participating partners, and are implemented directly by the individual components. The latter are deployed and coordinate themselves using shared tuplespace(s). Like our approach, the

tuplespace technology facilitates the execution of data-intensive workflows, since it allows for data distribution and yields a decrease of messages being passed between the interacting components. Unlike our approach, however, the use of tuplespace technology decouples data maintenance from the execution infrastructure. Although tuplespaces are proven technology and could be used as an alternative mechanism for data sharing by the BPELcube engine, their deployment would require additional hardware resources. Furthermore, the overall approach requires considerable preparatory work such as component configuration to be conducted at deployment time, which could eventually become a scalability bottleneck.

In order to effectively separate the concerns of regular BPEL engines and various other complex aspects, including decentralized orchestration, Jiménez-Peris et al. proposed the ZenFlow BPEL engine [24]. ZenFlow employs techniques from reflective and aspect-oriented programming, and makes use of specialized meta-objects to describe and plug the extensions necessary for decentralized execution into the centralized BPEL engine. In this work, however, decentralization is enabled by means of a cluster of centralized BPEL engines, with each one being responsible for the execution of the whole process each time. With BPELcube we follow a fine-grained decentralization strategy, whereby the BPEL process is decomposed into the consituent activities, the execution of which is distributed among the nodes of a P2P network.

The CEKK machine that was presented by Yu [8] supports P2P execution of BPEL processes based on the continuation-passing style. In this approach, the execution control is formalized as a continuation, and is passed along from one processing unit to another without the interference of a central coordinating component. In this distributed execution environment, special attention is paid to the problem of failure handling and recovery, while a number of extensions to the BPEL language are introduced. Overall, this approach focuses on the formalization of a distributed process execution model and does not address aspects related to the structure of the P2P infrastructure, or the distribution of process activities and variables. Furthermore, it lacks an evaluation that would allow us to assess its applicability to the execution of long-running and data-intensive BPEL processes.

3 The BPELcube Framework

We introduce BPELcube, a framework comprising a scalable architecture and a set of distributed algorithms to support the efficient execution of BPEL processes, which are long-running, consume and/or produce voluminous data, while they are simultaneously accessed by large numbers of clients. Central to our approach is its underlying P2P infrastructure, which implements a binary hypercube topology to organize an arbitrary number of available nodes.

In general, a complete binary hypercube consists of $N = 2^d$ nodes, where d is the number of dimensions equaling to the number of neighbors each node has. Hence the network diameter, i.e. the smallest number of hops connecting two most distant nodes in the topology, is $\Delta = \log_2 N$.

Hypercubes have been widely used in P2P computing [25] [26] [27], and are particularly known for a series of attributes, which are also fundamental for the applicability of our approach:

- *Network symmetry*. All nodes in a hypercube topology are equivalent. No node incorporates a more prominent position than the others, while any node is inherently allowed to issue a broadcast. Consequently, in our case, any node can become the entry point for the deployment and execution of a process.
- *Efficient Broadcasting*. It is guaranteed that, upon a broadcast, a total of exactly $N - 1$ messages are required to reach all N nodes in the hypercube network, with the last ones being reached after at most Δ steps, regardless of the broadcasting source. Since broadcasts are extensively used in our approach for the deployment and undeployment of BPEL processes, this property proves to be critical in terms of performance.
- *Cost-effectiveness*. The hypercube topology exhibits an $O(\log_2 N)$ complexity with respect to the messages that have to be sent, for a node to join or leave the network. Hence, the execution of the respective join and leave protocols does not inflict the overall performance of the distributed BPEL engine.
- *Churn resilience*. It is always possible for the hypercube topology to recover from sudden node losses. This makes possible the deployment of the distributed BPEL engine in less controlled WAN environments, if needed, where churn rates are naturally higher than the ones met in centrally administered LANs.

Each node in the hypercube topology is capable of executing one or more individual BPEL activities as part of a given process instance execution, while also maintaining one or more of the instance's data variables. Thus, one or more nodes are engaged to contribute in the execution of a process instance, and coordinate with each other in a completely decentralized manner that is exclusively driven by the defined structure of the corresponding process.

3.1 BPELcube Node Architecture

The main internal components of a node participating in the BPELcube engine are shown in Figure 2. The *P2P Connection Listener* acts as the entry point of each node accepting incoming requests from other nodes in the hypercube. Each request is bound to a new P2P connection, which is then passed to a *P2P Connection Handler* for further processing. Since the latter runs in a separate

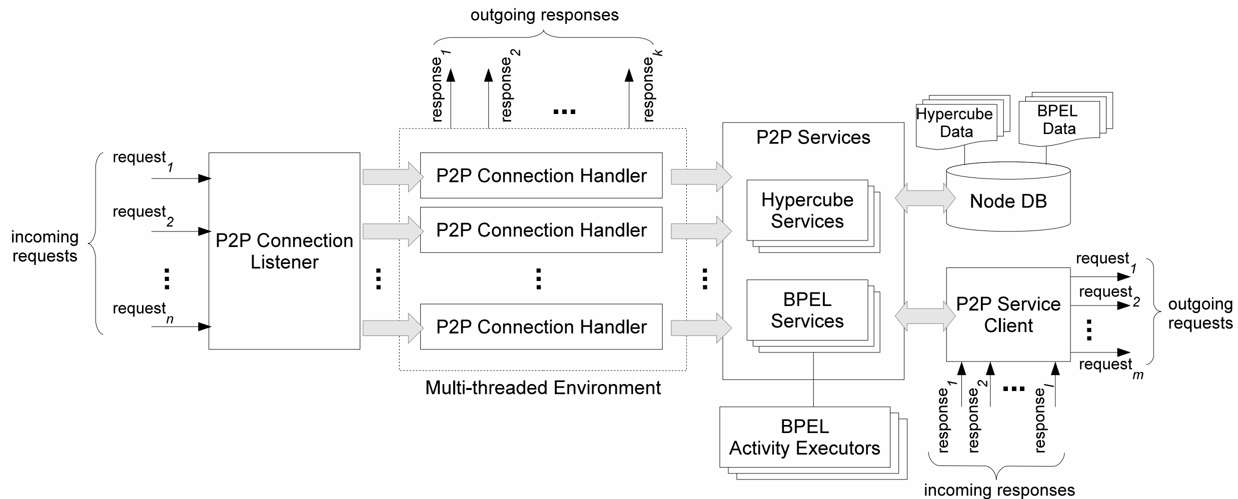

Fig. 2. Internal architecture and main components of the BPELcube node.

thread, it is possible for a node to simultaneously serve more than one incoming requests.

Depending on its type, a request is always associated with a particular *P2P service*, which the P2P Connection Handler selects, instantiates and executes. P2P services fall into two distinct categories:

- *Hypercube services* are used by the node to perform the various tasks needed for the maintenance of the hypercube topology. Such tasks implement the join and leave algorithms of the hypercube protocol, as well as additional functionality such as broadcasting, random walks, heartbeat, etc., which is essential for the P2P network.
- *BPEL services* encapsulate all functionality necessary for the distributed deployment, execution, and monitoring of BPEL processes by the BPELcube engine. Such services provide for the execution of individual BPEL activities (by employing the appropriate *BPEL activity executors*), the read/write of process variables, the response to notifications such as the completion of an activity or the completion of a process, etc.

The P2P services may follow a simple one-way communication, or otherwise implement the request-response pattern, in which case the corresponding P2P Connection Handler is used to send back the response message. The execution of most supported P2P services includes the invocation of one or more P2P services on other nodes within the hypercube. This is typical for instance in the hypercube service implementing the broadcast scheme, or the BPEL services that are used to execute a particular activity. To support such situations, each node is equipped with a *P2P Service Client*, which is responsible for establishing a P2P connection with a specified node, submitting the prepared service request, and receiving the corresponding response, if any. Finally, the majority of the supported P2P services make use of a local database that is embedded within the node. The database holds all information that is needed by a node to participate in the hypercube topology, and also maintains the various tuples, which are generated upon deployment and execution of a BPEL process. The BPEL data may be optionally replicated at the node's immediate neighbors, in order to improve the network resilience to higher churn rates. In such a case, to ensure data consistency, each change on the BPEL data of a node yields the exchange of a maximum of $log_2 N$ replication messages.

In the following paragraphs, we proceed with the description of our approach, explaining how the BPELcube architecture is used for the distributed deployment, execution, and monitoring of BPEL processes.

3.2 Process Deployment

For a BPEL process to be deployed to the BPELcube engine, a request containing a bundle with all necessary files needs to be submitted to one of the available nodes in the hypercube. In particular, this bundle contains the BPEL process specification, the WSDL interface, the WSDL files of all external services, as well as any potentially required XML schemas and/or XSLT transformation files. Upon receipt of the deploy request, the node first performs a syntactic validation of the included files, and then decomposes the process into its constituent activities and variables. The goal is to generate a convenient process representation that will facilitate its decentralized execution. Similar to the work reported in [19], our BPEL decomposition mechanism relies on the use of Program Dependence Graphs (PDGs) for representing the control, data, and synchronization dependencies of the process activities. From a well-formed PDG, it is then easy to decompose the original BPEL process (i.e. to identify the individual process activities and variables) by simply traversing the graph structure.

As soon as it has been properly parsed and decomposed, a BPEL process p is stored in a 4-tuple as follows:

$$\langle id_p, V, A, id_m \rangle \quad (1)$$

In this tuple, id_p is the unique identifier of the BPEL process, $V = \{id_{v_i}\}_{i=1}^{n}$ is a set containing the unique identifiers of all variables defined within the process, A is an ordered set containing all its activities in order of appearance, while id_m is the unique identifier of the *main* activity, i,e, the one that is always executed first.

Each element $a \in A$ represents an activity of the given process and takes the form of an 8-tuple

$$\langle id_a, s, id_{p(a)}, V_r, V_w, L_s, L_t, C \rangle \quad (2)$$

where:

- id_a is the unique identifier of the activity within the context of its owner process
- s holds the BPEL specification of the activity that will be used for its execution
- $id_{p(a)}$ is the unique identifier of the immediate parent activity of a, i.e. the structured activity that triggers the execution of a
- V_r is a set containing the identifiers of all variables that the activity reads from
- V_w is a set containing the identifiers of all variables that the activity writes to
- L_s is a set containing the identifiers of the activities which are synchronized with activity a and must be completed before a
- L_t is a set containing the identifiers of the activities which are synchronized with activity a and will wait for it to complete before executing themselves
- C is a set containing the identifiers of all activities, the parent of which is activity a

The deployer node persists the decomposed process locally, and initiates a broadcast within the hypercube, in order to efficiently send the process tuple to all available nodes. Thus, at the end of the broadcast, all nodes in the hypercube are in a position to serve requests for the execution of that particular process, without having to carry out the potentially time-consuming tasks of process validation and decomposition.

The undeployment of a BPEL process is performed in a similar way. It may be triggered by any node in the hypercube upon receipt of an undeploy request containing the process identifier. The recipient node checks if it is currently involved in the execution of one or more instances of that process. If it is, it schedules the deletion of the corresponding process tuple after all instances are completed. Otherwise, the process tuple is immediately removed from the node's local database. Then, the node initiates a broadcast within the hypercube sending the undeploy request to all available nodes. Each recipient performs the same algorithm in order to safely remove the process tuple from its local database without affecting any running instances of that process. Thus, the process tuple is eventually removed from all nodes in the hypercube and the undeployment is completed.

3.3 Process Execution

The execution of an already deployed process is triggered each time an *ExecuteProcess* request is received by one of the available nodes in the hypercube, containing the process identifier id_p and the initial input, if any. To ensure even distribution of workload, the recipient node starts a random walk within the hypercube by means of shortest-path routing, in order to appoint the node that will actually take over the role of the execution *manager*.

The appointed manager creates a new P2P session for the execution of the process, and stores it in a tuple:

$$\langle id_{p2p}, id_p, E_a, E_v, e_m, s_p \rangle \quad (3)$$

Each P2P session tuple is distinguished by a unique identifier id_{p2p}, while it is associated with the process to be executed through the process identifier id_p. In addition, it includes two tables, $E_a = \{(id_a, e)_i\}_{i=1}^{|A|}$ and $E_v = \{(id_v, e)_i\}_{i=1}^{|V|}$, which map the process activities and variables to the endpoint addresses of the *worker* nodes responsible for their execution and maintenance, respectively. Finally, the P2P session tuple contains the endpoint address e_m of the manager node, as well as the current status s_p of the process instance (i.e. *waiting*, *running*, *completed*, or *failed*).

3.3.1 Recruiting Worker Nodes

In order to properly fill in the tables E_a and E_v before starting out the execution of the process, a distributed recruitment algorithm is carried out. In doing so, the manager first assigns the initial *receive* and final *reply* activities to itself, and therefore takes over the responsibility for maintaining the variable holding the user input. Then, it broadcasts its new timestamp of last use to all its immediate neighbors in the hypercube. Afterwards, it selects its *Least Recently Used (LRU)* hypercube neighbor in the lowest possible dimension, in order to send to it a recruitment request containing the new P2P session tuple.

The recipient of a recruitment request continues the recruitment procedure by executing Algorithm 1. First, the receiver node updates the first entry in E_a where e is null with its own endpoint address, thereby appointing itself as a worker in the context of the particular P2P session. Then, it creates a new triple representing the specific activity instance:

$$\langle id_{p2p}, id_a, s_a \rangle \quad (4)$$

As it can be seen, the activity instance is identified by the combination of the P2P session and activity identifiers, and further has an execution status $s_a \in \{waiting, running, completed, failed\}$.

In case the assigned activity writes to a variable that has not yet been assigned to any other node (i.e. there is no entry for that variable in table E_v), the node also becomes responsible for the maintenance of that variable and updates E_v with a new entry accordingly.

Algorithm 1: Processing of a recruitment request.

```
    input: ⟨id_p2p, id_p, E_a, E_v, e_m⟩
 1  begin
 2      e_L ← get local endpoint address
 3      Get process tuple based on id_p
 4      recruitment_complete ← true
 5      foreach (id_a, e) ∈ E_a do
 6          if e = null then
 7              e ← e_L
 8              Update E_a
 9              recruitment_complete ← false
10              Get activity tuple based on process tuple
                and id_a
11              V_w ← get from activity tuple
12              foreach id_v ∈ V_w do
13                  (id_v, e) ← get corresponding entry
                    from E_v
14                  if e = null then
15                      e ← e_L
16                      Update E_v
17                  end
18              end
19              N ← get all hypercube neighbors
20              Broadcast timestamp of last use to all
                nodes ∈ N
21              break
22          end
23      end
24      if recruitment_complete = false then
25          w ← get LRU neighbor in lowest dimension
26          e ← get endpoint address from w
27          Send Recruitment request to e
28      else
29          Send RecruitmentCompleted notification to e_m
30      end
31  end
```

Each variable that a node becomes responsible for in the context of a P2P session is locally stored in a triple

$$\langle id_{p2p}, id_v, v \rangle \tag{5}$$

where id_{p2p} is the identifier of the P2P session, id_v is the identifier of the variable, and v is a holder of the variable value.

As soon as the aforementioned updates are performed, the node broadcasts its new timestamp of last use to all its immediate neighbors in the hypercube. Finally, the new worker selects its LRU hypercube neighbor in the lowest possible dimension and forwards to it the recruitment request which now contains the updated P2P session tuple.

The same steps are performed each time a node is visited during the recruitment procedure, until all entries in E_a are properly set, and all variables are assigned in E_v. Our algorithm ensures that the process activities and variables will be distributed to the least recently used nodes in a fair manner, although some nodes may be assigned more than one activities and/or variables, depending on the process at hand and the size of the hypercube. In any case, the last recruited node sends back to the manager a notification containing the updated P2P session tuple, thereby signaling the completion of the recruitment procedure. Finally, the manager sends a message to the client containing the P2P session tuple, so that the client can use the included information later for monitoring purposes.

It should be noted that, there is no one-to-one correspondence between the activities and the hypercube nodes. In other words, at any given time, it is possible for a node to be responsible for more than one activities belonging to more than one process instances. Depending on the available resources, the dimension of the hypercube may be scaled up in order to improve the performance of BPELcube as the number of concurrent process instances increases.

Example: Let us describe how the recruitment algorithm works in the case of the landslide BPEL process of Figure 1. For the sake of simplicity in our example, we assume that the BPELcube engine has just started and comprises a hypercube of eight nodes (3-cube). Figure 3, read from left to right and top to bottom, demonstrates the sequence in which the hypercube nodes are visited upon receipt of an execution request by node 000, while Table 1 shows the recruitment results, i.e. the distribution of the BPEL activities and variables to the hypercube nodes. As it can be seen, the recruitment algorithm managed to engage all available nodes while taking into account their frequence of use upon distribution of the workload.

3.3.2 Running the Process Instance

As soon as the recruitment of worker nodes is finished, the manager node retrieves from table E_a the endpoint address of the worker responsible for the main activity of the process and sends to it an *ExecuteActivity* request containing the activity identifier and the P2P session tuple that corresponds to the process instance. Upon receipt of the request, the worker node executes the specified activity according to Algorithm 2, as described below.

First, the worker node uses the process identifier id_p that is included in the P2P session tuple in order to retrieve the corresponding process tuple from its local database. Then, it uses the activity identifier id_a to extract the corresponding activity tuple from set A.

Before proceeding with the activity execution, the worker waits until an *ActivityCompleted* notification is sent by all activities in L_s, in the context of the same P2P session. Afterwards, the worker uses the contents of V_r in order to resolve the variables that the activity needs to read from, and retrieves the endpoint addresses of the nodes responsible for their maintenance in the P2P session. Then, for each variable, it sends a *ReadVariable* request to the respective node and caches the response, or simply reads the variable locally, in case the worker is the actual holder.

As soon as all needed variables are read, the worker runs the activity and stores the values of all variables found in V_w. This is either done locally, in case the

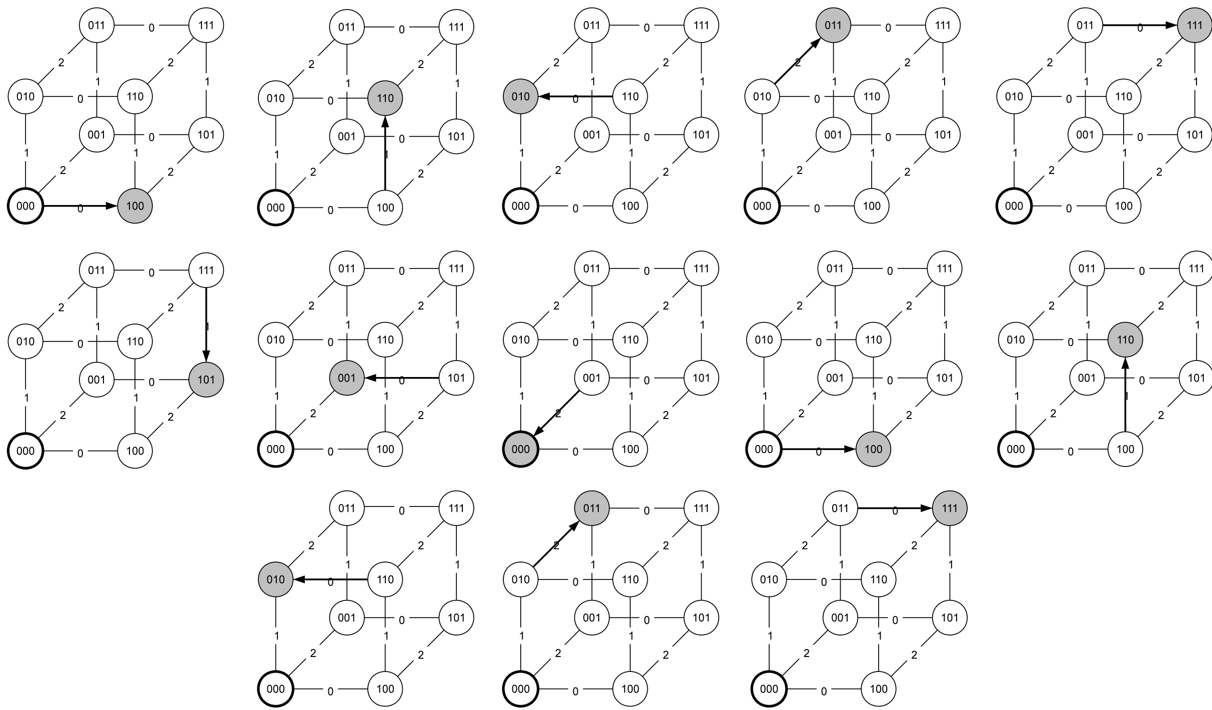

Fig. 3. Recruitment of workers for the execution of the landslide environmental model.

TABLE 1
Recruitment procedure results.

Hypercube Node	Assigned Activities	Assigned Variables
000	Receive, Reply, Invoke2	LandslideInput, Precipitation
100	Sequence1, Assign3	HydroModelInput
110	Flow, Invoke3	HydrologicalModel
010	Sequence2, Assign4	SafetyFactorsMapInput
011	Assign1, Invoke4	DEMInput, SafetyFactorsMapOutput
111	Invoke1, Assign5	DEM, LandslideOutput
101	Sequence3	–
001	Assign2	PrecipitationInput

variable has been assigned to that worker, or remotely, by sending a *WriteVariable* request to the respective node.

Finally, the worker sends an *ActivityCompleted* notification to the nodes in charge of all activities in L_t, containing the activity identifier and the updated P2P session tuple. In case the completed activity has a parent, the worker also sends the same *ActivityCompleted* notification to the node in charge, otherwise it sends a *ProcessCompleted* notification to the manager, signaling the completion of the process instance.

In the context of a process instance, the same algorithm is repeated by each node that receives an *ExecuteActivity* request. In this way, the process activities are executed one after the other, according to the designated control flow. The proper execution order of all nested activities is ensured by the BPEL activity executors that implement their parent structured activities.

Example: Let us describe how the structured activity *Sequence2* of the landslide BPEL process will be executed based on the results of the recruitment procedure shown in Table 1. In principle, a *sequence* activity within a BPEL process is responsible for sequentially executing all its child activities. In our example, node 010, which is responsible for the execution of *Sequence2*, sends an *ExecuteActivity* request to node 011, and waits until it receives back an *ActivityCompleted* notification. Node 011 is responsible for the execution of activity *Assign1*, which is the first child activity of *Sequence2*.

Since the activity *Assign1* is synchronized with activity *Assign2* through a BPEL link (see arrow in Figure 1), node 011 will wait until an *ActivityCompleted* notification is sent from node 001. Then, it sends a *ReadVariable* request to node 000, in order to retrieve the value of the *LandslideInput* variable. After that, the node proceeds with the execution of the copy statements within the *assign* activity, and locally writes the produced outcome to the *DEMInput* variable. At this point, the *Assign1* activity has completed and node 011 sends an *ActivityCompleted* notification to node 010, which is in charge of the parent activity, *Sequence2*.

Algorithm 2: BPEL activity execution.

```
input: id_a, ⟨id_p2p, id_p, E_a, E_v, e_m⟩
1  begin
2      e_L ← get local endpoint address
3      Get process tuple based on id_p
4      Get activity tuple based on process tuple and id_a
5      Wait for all activities in L_s to complete
6      foreach id_v ∈ V_r do
7          e ← get endpoint address from E_v for id_v
8          if e = e_L then
9              Read value of variable id_v locally
10         else
11             Read value of variable id_v from e
12         end
13     end
14     Execute activity a
15     foreach id_v ∈ V_w do
16         e ← get endpoint address from E_v for id_v
17         if e = e_L then
18             Persist new value of variable id_v locally
19         else
20             Persist new value of id_v in e
21         end
22     end
23     Notify all activities in L_t
24     if p(a) ≠ null then
25         e ← get endpoint address from E_a for p(a)
26         Send ActivityCompleted notification to e
27     else
28         Send ProcessCompleted notification to e_m
29     end
30 end
```

Node 010 resumes the execution of *Sequence2*, which dictates that the *Invoke1* activity be executed next. To do so, an *ExecuteActivity* request is sent to node 111 which is responsible for that activity. Before performing the actual invocation of the Digital Elevation Model WCS, node 111 retrieves the required input by reading the *DEMInput* variable from node 011. After invocation, the service output is locally written to variable *DEM*, and an *ActivityCompleted* notification is sent back to node 010, allowing it to complete the execution of activity *Sequence2*, and send the appropriate notification to node 110, which is in charge of the execution of the parent *Flow* activity.

3.4 Process Monitoring

The ability to monitor various aspects of a long-running process instance while executing is a critical user requirement in most application domains. The BPELcube engine addresses this need and facilitates the respective monitoring tasks. At any time during execution of a process instance, the client can easily obtain monitoring details in a non-intrusive manner. Typical monitoring information includes the process instance status, the status of individual activities, as well as the current values of the process variables. In the following, we describe how the provided monitoring mechanism enables the retrieval of such details, which can be presented to the client in a user-friendly, visualized manner through appropriate front-end software.

3.4.1 Retrieving Process Instance Status

The status of a process instance can be obtained directly from its appointed manager node, the endpoint address of which is known to the client through the P2P session tuple. In doing so, the client sends a *GetProcessInstanceStatus* request containing the identifier id_{p2p} of the corresponding P2P session. In response, the manager retrieves the P2P session tuple and sends back to the client the process execution status value s_p.

3.4.2 Retrieving Individual Activity Status

While waiting for a long-running process instance to complete, the client often needs to keep track of the execution progress at the activity level. Our monitoring mechanism enables the client to retrieve this information by directly contacting the nodes responsible for the execution of each individual activity. More specifically, for a given process instance and activity, the client resolves from the P2P session tuple the endpoint address of the corresponding node through table E_a, and submits to it a *GetActivityStatus* request containing the P2P session identifier id_{p2p} and the activity identifier id_a. The node then simply returns to the client the current status s_a of the specified activity instance taken from the corresponding activity instance tuple.

3.4.3 Retrieving Intermediate Results

BPEL processes usually depend on data from many external sources, which are commonly retrieved through invocations of the appropriate Web services, and are stored in the various variables. In many applications, those intermediate data may be equally important to the user as the final result. Our monitoring mechanism facilitates the extraction of intermediate results from the 'black box' surrounding the execution of a BPEL process. Thanks to the information stored in table E_v in each P2P session (i.e. process instance), the client is able to directly interact with the node in charge of each variable, and retrieve the current value of the specified variable by sending a *ReadVariable* request containing the P2P session and variable identifiers.

3.5 Resilience to Network Failures

Thanks to the hypercube topology, the BPELcube engine exhibits resilience to changes in the underlying P2P network. In general, when a node in the hypercube goes offline, we are interested in preserving all of its hypercube-specific and BPEL-specific data. The latter are translated into the variable values, activity assignments, and P2P session tuples that are maintained by the node. Here we define two recovery strategies corresponding to graceful and unexpected node departures.

3.5.1 Recovery From Graceful Node Departures

In the case of a graceful node departure from the hypercube, the respective leave protocol is executed and one of its neighbors takes over its position. By properly extending the hypercube leave protocol we ensure that, in addition to the hypercube-specific information, the replacing node will also obtain all BPEL-specific data from the departing node. After that, the replacing node broadcasts a notification, so that all the other nodes in the hypercube update the activity and variable assignments in all affected P2P session tuples. This is simply done by replacing the departed node's endpoint address with that of the replacing node.

3.5.2 Recovery From Unexpected Node Departures

If the hypercube topology has been deployed to an open environment with high churn rate, it is always possible that nodes go unexpectedly offline, without performing the hypercube leave protocol. In such cases, the hypercube protocols ensure that the vacant positions will be taken over by other nodes eventually, leading to a stable and consistent topology. However, the only way to also ensure that the BPEL-specific data of the departed nodes will be preserved is to enable replication. This feature is optional in BPELcube, mainly because it comes with the price of increased network traffic and resources consumption. Nevertheless, thanks to the hypercube topology, we are able to keep the side-effects limited, as the BPEL-specific contents of each node are replicated only to its neighbors yielding the exchange of a maximum of $\log_2 N$ additional messages each time a change is made in the node's BPEL data.

4 Evaluation

With the use of a prototype implementation in Java, we performed an evaluation of BPELcube in a number of experiments. The goal was to quantitatively assess the benefits of the proposed architecture and algorithms in terms of average process execution time and throughput. In order to cover various aspects affecting the performance, the experimental measurements were taken in three different settings. More specifically, we investigated the effect of (i) the request rate, (ii) the message sizes, and (iii) the external service delays on the performance of the distributed execution engine. Additionally, in all cases, by comparing BPELcube to centralized and clustered architectures, we identified their limitations and gained insight on the conditions where each approach would be best suited.

4.1 Experimental Setup

For our experiments we utilized a 100Mbps LAN of 16 workstations, with each one running Linux Ubuntu 8.0.4 (kernel 2.6.24-31) on Intel (R) Core (TM) 2 Duo CPU E6750@2.66GHz processor and 1 GB RAM. In this setting, we deployed a single node so as to use it as the centralized BPEL execution engine, a cluster of two nodes with a simple load balancer acting as a clustered BPEL engine, and a three-dimensional hypercube (3-cube) of 8 nodes that formed our distributed BPEL engine. Another node was reserved for the deployment of the client, which we implemented and used to feed the aforementioned nodes with ExecuteProcess requests at various rates.

All measurements were taken by executing the BPEL process of Figure 1. In order to be able to modify and test with different message sizes and delays, we implemented simulators of the four actual Web services being invoked by the process, and deployed them on the same LAN. Finally, the duration of the client execution for the retrieval of each measurement in all experiments was set to 10 minutes.

4.2 Experimental Results

4.2.1 Varying Request Rate

In the first part of our experiments, we examined the behavior of the three different architectures in the presence of up to 300 concurrently running instances of the landslide process. The message size was set at 1024kb, while the external service delay was set at 15 seconds.

As the results in Figure 4 suggest, all approaches exhibited similar throughput and execution times for rates of up to 70 requests per minute. However, the performance of the centralized deployment was seriously degraded for larger request rates, making this approach not able to serve more than 150 requests per minute. It is also worth mentioning that, in the case of 150 requests per minute, the average process execution time was about five times longer than that of BPELcube.

Besides, even though the clustered architecture was responsive at higher request rates, its throughput was considerably deteriorated, while the average process execution times were significantly increased. Concluding, the measurements made clear that this approach cannot support more than 200 concurrent requests with acceptable response times. On the other hand, our distributed architecture managed a relatively steady throughput regardless of the request rate, and a slight increase on the average process execution times.

4.2.2 Varying Message Size

In the second part of the experiments, we assessed the capability of BPELcube to cope with larger data produced by the external Web services, and also identified the limitations of the centralized and clustered deployments. Like in the previous set of runs, the external service delay was again specified at 15 seconds, while we configured the client to feed the engine with 50 *ExecuteProcess* requests per minute.

The measured throughput and average process execution times are shown in Figure 5. The results showed that, for message sizes up to 1280kb, all approaches perform in a similar way. The performance of the centralized

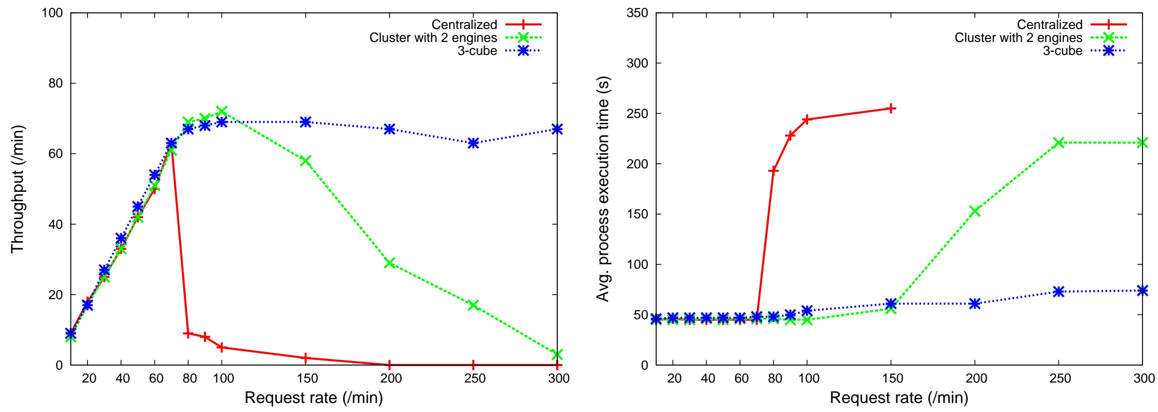

Fig. 4. Performance measurements with varying request rates.

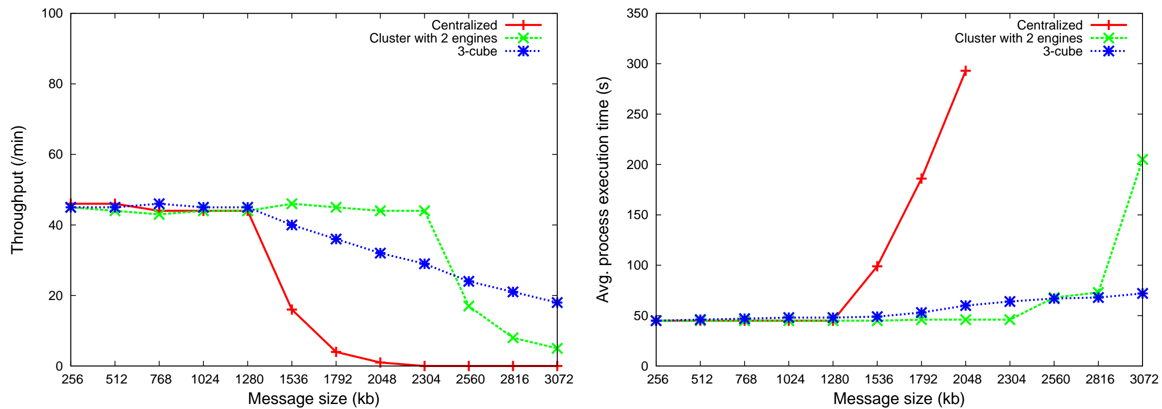

Fig. 5. Performance measurements with varying message sizes.

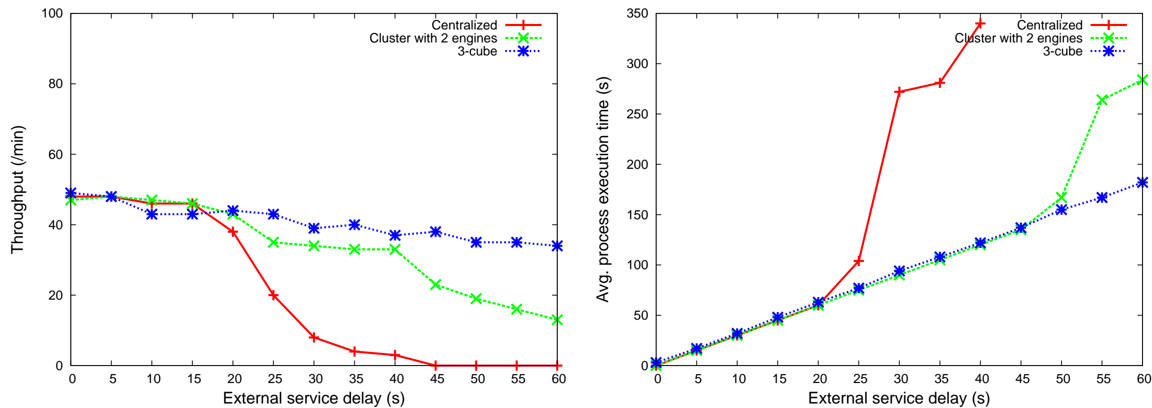

Fig. 6. Performance measurements with varying external service delays.

engine was seriously degraded for larger data, as it was practically stalled for message sizes larger than 2304kb. On the other hand, the clustered engine exhibited the best performance of all approaches for message sizes up to 2304kb, while our distributed engine was able to maintain a relatively steady average process execution time and a slowly decreasing throughput in all cases.

The partial superiority of the clustered BPEL engine over BPELcube was due to the significantly smaller number of required message exchanges, as the produced data were maintained in two nodes only. However, the experiment showed that the cluster's resources capacity was not adequate to accommodate processes involving variables with size larger than 2560kb. On the contrary, such issues were not observed in our approach, thanks to the distribution of the process variables in all nodes available in the hypercube.

All in all, the results of this part of the experiments could provide a guide for the selection of the most appropriate architecture in terms of scalability. In other

words, the deployment of a cluster could be avoided for processes with small-sized variables, whereas the use of a distributed architecture such as the one championed by BPELcube would be required to accommodate concurrently running processes with data variables of higher volume.

4.2.3 Varying External Service Delay

The third part of the experiments concerned the implications of external service delays to the performance of the BPEL execution engine, in the presence of multiple concurrently running process instances. To investigate such situations, we programmatically enforced an artificial delay in the execution of the four Web services being invoked by the landslide process that ranged between zero and 60 seconds. Furthermore, we set a constant message size of 1024kb and ensured a steady rate of 50 *ExecuteProcess* requests per minute coming from the client.

The results, shown in Figure 6, indicate once again the limitations of centralized process execution, as the centralized engine was again stalled for service delays lasting more than 45 seconds. This can be explained by the fact that too many process instances remained active, as each one of them required more time to complete due to the extended waiting for responses of the invoked services. Besides, although the clustered engine was able to cope with the multiple concurrent and long-running process instances, its throughput was up to 50% lower than that of BPELcube. Regarding the average process execution times, both the centralized and clustered approaches were outperformed by our distributed BPEL engine for delays of more than 50 seconds.

4.2.4 Workload Distribution

We conclude the presentation and discussion of our experimental results with an evaluation of BPELcube in terms of workload distribution. In the context of BPEL process execution, workload amounts to the total number of process activities and variables that are assigned to each hypercube node over time. In order to assess the efficiency of the distributed hypercube-based architecture and algorithms, we run another 10-minute round of feeding the deployed three-dimensional hypercube with landslide process execution requests. The request rate was set at 50/min, the message size was specified at 1024kb, while the external service delay was set at 15 seconds.

As it can be seen in Figure 7, our recruitment algorithm achieved a remarkably even distribution of the activities, exploiting all the available hypercube nodes and considering at the same time their frequence of use. On the other hand, the measurements revealed some deviations regarding the distribution of process variables. Nevertheless, such behavior was anticipated, since apart from the *receive*, *assign*, and *invoke* activities, the rest of the activities found in the landslide BPEL process do not produce any data (i.e. they do not write to any of the process variables). Hence, the same nodes responsible for the execution of data-producing activities were inevitably also used for the maintenance of the corresponding variables that held the produced data.

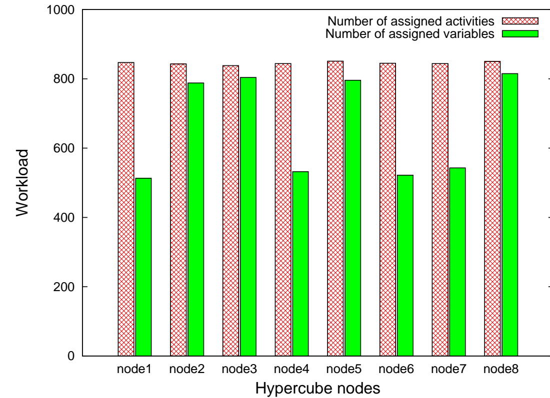

Fig. 7. Distribution of activities and variables across the nodes of the three-dimensional hypercube BPEL engine.

5 Discussion

We presented a distributed architecture based on the hypercube P2P topology along with a set of algorithms that enable the decentralized execution of BPEL processes. Our approach targets towards the improvement of the average process execution times and the enhancement of the overall throughput of the execution infrastructure, in the presence of multiple long-running process instances that involve the exchange of large data. The presented algorithms support the decomposition of a given BPEL process and the subsequent assignment of the constituent activities and data variables to the available hypercube nodes. Execution is then performed in a completely decentralized manner without the existence of a central coordinator. Our distributed approach also provides a lightweight monitoring mechanism that does not intrude into the process execution, but rather allows the retrieval of monitoring information from the hypercube in a seamless manner.

We evaluated our approach in a series of experiments, and compared it with centralized and clustered architectures in terms of performance. The retrieved measurements indicate that our hypercube-based architecture is more suitable for the execution of long-running and data-intensive processes, while it is able to accommodate more concurrent clients than the other two architectures. Moreover, thanks to the even distribution of workload, our approach copes with large data in a more efficient manner.

In future work, we aim to expand our worker recruitment algorithm so as to consider additional factors like network proximity or other QoS, which are complementary to the frequence of use of the employed nodes. This expansion will facilitate the deployment of the hypercube-based engine on less controlled settings

such as WAN networks. We are also interested in extending the proposed architecture to support Cloud-based deployment of the BPELcube engine. We anticipate that by moving BPELcube to the Cloud, we will be able to exploit elasticity capabilities for dynamically increasing or decreasing the hypercube dimension. This way, the BPELcube engine will be able to effectively and timely respond to workload changes. Finally, in terms of implementation, we will investigate the use of parallel query processing techniques to further enhance the performance of BPELcube nodes, in the presence of multiple concurrently running process instances.

REFERENCES

[1] OASIS, "Web Services Business Process Execution Language Version 2.0," Apr. 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[2] B. Schäffer and T. Foerster, "A client for distributed geo-processing and workflow design." *Journal of Location Based Services*, vol. 2, no. 3, pp. 194–210, 2008.

[3] A. Weiser and A. Zipf, "Web service orchestration of ogc web services for disaster management," in *Geomatics Solutions for Disaster Management*, ser. Lecture Notes in Geoinformation and Cartography, J. Li, S. Zlatanova, A. G. Fabbri, W. Cartwright, G. Gartner, L. Meng, and M. P. Peterson, Eds. Springer Berlin Heidelberg, 2007, pp. 239–254.

[4] X. Meng, F. Bian, and Y. Xie, "Research and realization of geospatial information service orchestration based on BPEL." in *Proceedings of the 2009 International Conference on Environmental Science and Information Application Technology, ESIAT 2009*. IEEE Computer Society, 2009, pp. 642–645.

[5] F. Theisselmann, D. Dransch, and S. Haubrock, "Service-oriented architecture for environmental modelling - the case of a distributed dike breach information system," in *Proceedings of the 18th World IMACS/MODSIM Congress*, 13-17 July 2009, pp. 938–944.

[6] D. Roman, S. Schade, A. J. Berre *et al.*, "Environmental services infrastructure with ontologies - a decision support framework," in *Proceedings of EnviroInfo 2009: Environmental Informatics and Industrial Environmental Protection: Concepts, Methods and Tools*. Shaker Verlag, 2009, pp. 307–315.

[7] G. Li, V. Muthusamy, and H.-A. Jacobsen, "A distributed service-oriented architecture for business process execution." *ACM Transactions on the Web*, vol. 4, no. 1, 2010.

[8] W. Yu, "Peer-to-peer execution of bpel processes." in *CAiSE Forum*, ser. CEUR Workshop Proceedings, J. Eder, S. L. Tomassen, A. L. Opdahl, and G. Sindre, Eds., vol. 247. CEUR-WS.org, 2007.

[9] M. Armbrust, A. Fox *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[10] P. Missier, S. Soiland-Reyes *et al.*, "Taverna, reloaded," in *SSDBM 2010*, M. Gertz, T. Hey, and B. Ludaescher, Eds., Heidelberg, Germany, June 2010.

[11] S. Callaghan, E. Deelman *et al.*, "Scaling up workflow-based applications," *J. Comput. Syst. Sci.*, vol. 76, no. 6, pp. 428–446, Sep. 2010.

[12] I. Altintas, C. Berkley *et al.*, "Kepler: An extensible system for design and execution of scientific workflows." in *SSDBM*. IEEE Computer Society, 2004, pp. 423–424.

[13] I. J. Taylor, M. S. Shields *et al.*, "Distributed P2P Computing within Triana: A Galaxy Visualization Test Case." in *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*. IEEE Computer Society, 2003, pp. 16–27.

[14] E. Deelman, G. Singh *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[15] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the kepler scientific workflow system," in *Proceedings of the 2006 international conference on Provenance and Annotation of Data*, ser. IPAW'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 118–132.

[16] W. Allcock, J. Bresnahan *et al.*, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 54–.

[17] J. Yan, Y. Yang, and G. K. Raikundalia, "SwinDeW-a p2p-based decentralized workflow management system." *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 36, no. 5, pp. 922–935, 2006.

[18] L. Gong, "JXTA: A Network Programming Environment," *IEEE Internet Computing*, vol. 5, no. 3, pp. 88–95, 2001.

[19] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing execution of composite web services." in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '04*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 170–187.

[20] L. Baresi, A. Maurino, and S. Modafferi, "Towards distributed BPEL orchestrations." *Electronic Communications of the EASST*, vol. 3, 2006.

[21] U. Yildiz and C. Godart, "Towards decentralized service orchestrations," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2007, pp. 1662–1666.

[22] L. Ai, M. Tang, and C. Fidge, "Partitioning composite web services for decentralized execution using a genetic algorithm," *Future Generation Computer Systems*, vol. 27, no. 2, pp. 157 – 172, 2011.

[23] D. Wutke, D. Martin, and F. Leymann, "Model and infrastructure for decentralized workflow enactment," in *Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08*. New York, NY, USA: ACM, 2008, pp. 90–94.

[24] R. Jiménez-Peris, M. Patiño Martínez, and E. Martel-Jordán, "Decentralized web service orchestration: A reflective approach." in *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*. ACM, 2008, pp. 494–498.

[25] M. T. Schlosser *et al.*, "Hypercup - hypercubes, ontologies, and efficient search on peer-to-peer networks." in *Proceedings of the First International Conference on Agents and Peer-to-Peer Computing, AP2PC 2002*, ser. Lecture Notes in Computer Science, G. Moro and M. Koubarakis, Eds., vol. 2530. Springer, 2002, pp. 112–124.

[26] H. Ren, Z. Wang, and Z. Liu, "A hyper-cube based p2p information service for data grid." in *Proceedings of the Fifth International Conference on Grid and Cooperative Computing, GCC 2006*. IEEE Computer Society, 2006, pp. 508–513.

[27] E. Anceaume, R. Ludinard *et al.*, "Peercube: A hypercube-based p2p overlay robust against collusion and churn." in *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008*, S. A. Brueckner, P. Robertson, and U. Bellur, Eds. IEEE Computer Society, 2008, pp. 15–24.

Michael Pantazoglou is a post-doc research associate at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, Greece. He is currently a member of the S^3Lab (http://s3lab.di.uoa.gr) group. His current research interests span service-oriented computing, with a focus on service discovery and composition, P2P computing, and semantic web technologies.

Ioannis Pogkas received his BSc and MSc from the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, Greece. His research interests focus on distributed search, reputation and execution mechanisms in peer-to-peer networks. He is also interested in mobile computing.

Aphrodite Tsalgatidou is associate professor at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, Greece. Aphrodite is the director of the S^3Lab group (http://s3lab.di.uoa.gr), which pursues research in service engineering, software engineering and software development.