Shepherd: Node Monitors for Fault-Tolerant Distributed Process Execution in OSIRIS *

Diego Milano
Department of Computer Sciences
University of Basel, Switzerland
diego.milano@unibas.ch

Nenad Stojnić
Department of Computer Sciences
University of Basel, Switzerland
nenad.stojnic@unibas.ch

ABSTRACT

OSIRIS is a middleware for the composition and orchestration of distributed web services that follows a P2P decentralized approach to process execution, providing already some degree of resilience to faults and high performance in large-scale computational clusters. In this paper, we present on-going work aimed at improving OSIRIS' fault tolerance capabilities. We introduce in OSIRIS new architectural elements for the maintenance of a virtual stable storage and the monitoring of activities of service instances, together with algorithms that allow execution to survive also failures that the system is currently not able to cope with.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications, cloud computing, grid computing

General Terms

Algorithms, Reliability

Keywords

Decentralized process execution, fault-tolerance, OSIRIS, monitoring, DHT

1. INTRODUCTION

OSIRIS (Open Service Infrastructure for Reliable and Integrated process Support) [16, 17] is a web service oriented middleware for the decentralized execution of distributed processes. These processes can be imagined as programs that coordinate the invocation of distributed web services. In OSIRIS, the orchestration of processes follows a fully distributed P2P approach that avoids single-points of failure, and therefore provides already some degree of resilience to failure. The approach also distributes the load related to process orchestration, avoiding the creation of bottlenecks and enabling the system to scale to a virtually unlimited number of nodes. Altogether, these features make OSIRIS suitable for the management of services in very large computational clusters.

Key features of OSIRIS are its approach to distributed computation based on the *programming in the large* paradigm, its decentralized approach to process execution supported by transactional guarantees, as well its self-healing and self-adaptation properties. Late binding of service instances, in conjunction with load balancing strategies, guarantee that the workload is economically spread on available computational resources. Furthermore, the execution of individual process instances is resilient to a wide class of network and node failures. Currently, the system is completely resilient to temporary node failures. If nodes are temporarily unavailable, the system continues to work properly. Also, thanks to late binding, permanent failures of nodes participating to the execution of a process instance, but not involved in a computation at the moment of failure, do not affect the execution.

However, the system cannot currently recover from permanent failures of nodes actively involved in a processing step. This kind of faults lead to the death of a process instance. These cases can be detected by clients that have launched the execution using timeouts, but there is no way to identify precisely at which point the process encountered a problem, or to recover the results of the partial computation that might have been already performed.

In this paper, we present on-going work on *Shepherd*, a mechanism aimed at improving OSIRIS' fault tolerance capabilities, in particular widening the class of faults from which the system can recover to include all cases of permanent node failures.

The solution adopted in this paper is based on the concept of "node monitor". Each node is assigned a supervisor by the system, that is responsible for monitoring the completion of tasks by the node. If a node fails during the execution of a given task, the monitor takes care of transferring the execution of the task to an alternative node. Permanently failed nodes that come back and find themselves already replaced with another node, are terminated in order to prevent inconsistent process execution. As monitors themselves may reside on unreliable nodes within the cluster, they are organized in a monitoring overlay in which more than one monitor is responsible for supervising a given node. If the current monitor of a node fails, another monitor is elected to continue the monitoring task.

*This work was funded by the SNF within the scope of the project SOSOA

WEWST '10, December 1, 2010, Ayia Napa, Cyprus

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 introduces basic notions regarding process execution in OSIRIS and discusses the current limitations of the framework. Section 4 presents our monitoring approach for failure handling in OSIRIS. More details about the organization of monitors are given in Section 5. Finally, Section 6 concludes this paper and gives a prospect to the future work.

2. RELATED WORK

There is a great number of approaches in the field of decentralized process execution, all of them aiming to cope with the shortcomings of the classic centralized approach. Initially, most of them were either based on concepts such as pre-instantiation of processes [21, 1] or central meta-data repositories [16, 17]. In the pre-instantiation approach, resources and control are pre-allocated in the distributed environment according to a distribution plan generated from static specifications of the processes. On the other hand, central meta-data repositories are used to provide dynamic runtime information to all involved peers in the environment. However, those early approaches suffer from some downsides. The pre-instantiation approach creates runtime unadaptability and unnecessary overhead, as not all pre-allocated resources are actually used. Systems with centralized meta-data repositories provide only partial decentralization of execution, and these repositories still represent *single points of failure* and might act as *bottlenecks*.

Recently, approaches based on mobile agents [20, 13] and continuation-passing messaging [23, 22, 24], aimed at tackling the above stated decentralized execution challenges, have attracted much attention among researchers. They both use the same basic principle of migrating execution from one peer to another and carrying along a payload of execution code, state and control. This model inverts the usual approach to distributed computation, as executable code is shipped to the nodes where the data of interest resides, and not the other way round. These approaches differ from that taken in OSIRIS, as in OSIRIS no code, apart from the process control flow definition, is transferred during execution. Approaches that come closer to a distributed orchestration, rely on a pre-instantiation approach. e.g. [11]

An important topic closely related to decentralized process execution is fault tolerance. The existing approaches for fault-tolerance can be broadly classified into two classes, *replicating* or *checkpointing* ([14]). *Replication* schemes involve replication of execution state on different sites in the environment and expensive agreement protocols for the selection of execution peers. This scheme raises issues such as consistency of replicas, which results in increased networking overhead and some degree of pre-instantiation. The *checkpointing* scheme involves the replication of execution state into stable storage for potential failure recovery. This scheme does not have the weaknesses of the previous one, but comes again with a price, as excessive checkpointing degrades performance, whereas deficient checkpointing results in an expensive recovery overhead.

An example based on mobile agent approach that uses replication is [20]. This solution suffers from the above described replication scheme weaknesses. As OSIRIS performs late binding, it does not require expensive protocols to elect responsibles for the execution. The approach presented in [23] uses a central "scoping" node which is responsible for execution control and for fault tolerance. The scoping node is assumed to be fault-tolerant. The solution we present in this paper also relies on monitoring, but uses a decentralized monitoring overlay of nodes which incorporates internal failure tolerance mechanisms.

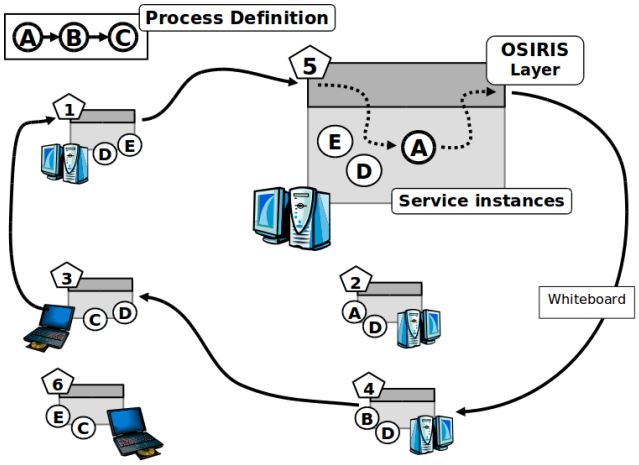

Figure 1: Process Execution in OSIRIS

Finally, a few proposals deal with the flexible execution of processes in large scale distributed environments like the grid. Some of these approaches rely on concepts anaologus to those found in OSIRIS, like late binding. However, most of them deal with incorporating traditional (centralized) workflow orchestration into grid environments,e.g. [4, 3], and are thus far from the distributed orchestration approach taken by OSIRIS.

3. PROCESS EXECUTION IN OSIRIS

In this section, we describe the regular process execution steps of OSIRIS, describing in particular how faults are handled and identifying cases that the current system is not able to cope with.

3.1 Peer-to-peer Process Execution

Applications built on top of OSIRIS can be described as processes, i.e. as programs orchestrating the invocation of distributed web services.

A process description encompasses a set of *activities*, each activity corresponding to the invocation of a service, and describes the control and data flow between them. OSIRIS process descriptions are abstract, they only specify the type or class of services to be invoked. The actual binding of services to concrete instances is decided entirely by the system at run-time (*late binding*), depending on the current configuration of the system (availability, load, cost of invocation). Following the paradigm of programming-in-the-large, processes themselves are wrapped by service interfaces, and can be invoked from within other processes.

The process description language used in OSIRIS allows for loops and conditional execution. Intra-process parallelization (forking and joining branches) is also possible.

The data flow is defined as a sequence of mappings from a data space into the process instance, called *whiteboard*, to the service request parameters, and back from the response values to the whiteboard after execution of an activity.

Process execution proceeds in a purely decentralized P2P

fashion. Nodes participating in an OSIRIS system deploy a thin software layer that provides local functionality for process instance navigation and routing. After the completion of the activity, the node *migrates* the control for process execution to one or more successor nodes by delivering an *migration token* containing flow-control information and the whiteboard.

To ensure that the data transfer is properly completed, a 2PC protocol [2] is applied between the two nodes.

In order for the execution to be completely decentralized, a certain amount of knowledge must be available to all involved nodes. This includes information used for routing (e.g. addresses of other nodes) and for load balancing (workload of other nodes). At the current state of its development, OSIRIS runs a few centralized system repositories for meta-data management. These repositories gather and distribute information through a publish-subscribe mechanism that minimizes the amount of meta-data exchanged between the central repositories and OSIRIS nodes [16]. This substantially eliminates the need for nodes to access the repositories during process execution. More details, including a performance evaluation of the scalability of the approach are given in [17].

A simplified example of process execution is shown at the top of Figure 1. The picture shows a set of network nodes (numbered 1 to 6), each equipped with an OSIRIS layer and each hosting a set of service instances of type A-E. The process definition at the top of the picture contains a sequence of three activities, corresponding to the invocation of services of type A, B and C respectively.

Suppose the process is initiated at node no. 1. According to the process description, the OSIRIS layer on that node must route the process to a node hosting a service instance of type **A**. This is the case for nodes no. 2 and no. 5. The precise node to which the process execution is forwarded, is determined at run-time (late binding) based on current load information. For our example, we assume that node no. 5 has been chosen. Node no. 1 thus migrates the execution to node no. 5, exchanging with it a migration token that contains control information and the whiteboard. On node no. 5, the OSIRIS layer executes the activity **A** by invoking the appropriate service instance running on the node. After the service instance has successfully terminated its execution, the OSIRIS layer of the node must again pick a node able to execute the next activity and route the execution to it. This procedure is repeated until the last activity of the process is executed. In our example, the execution migrates from node no. 5 to node no. 4 (executing an activity of type B) to node no. 3 (executing an activity of type C). The OSIRIS layer at the node executing the last activity will transfer the results to the node that initiated the process (node no. 1).

3.2 Failure Handling in OSIRIS

The OSIRIS framework is designed to be run in large computational clusters. In such environments, it is necessary to properly handle failures deriving from causes like hardware malfunctions (e.g. power loss or breakage of electronic components), network disruptions (e.g. link breakdowns) and software errors (e.g. bugs, memory leaks).

Example 1 does not explain the behavior of the system in case of faults of nodes involved in the process execution. Let us address the possible cases. If a node that already migrated the process state after executing an activity crashes, no special handling is needed, as the responsibility of the node within the process execution is limited to that activity. Also if one of the nodes that could be chosen to continue execution crashes, no special handling of the fault is involved. As OSIRIS relies on late binding, another node will be chosen for the migration. Another node will be chosen also if the target node crashes during the migration, as the system employs a 2PC transactional protocol, and a migration aborts if it is not completed correctly.

What happens if it is instead the node currently in charge for execution to fail? If the node becomes temporarily disconnected from the network, the system is still able to recover. In this case, the node will keep retrying to pass on the results of the computation until it succeeds. When the network problem disappears, the execution will proceed. The process execution survives also if the node crashes, but later recovers (e.g. it is temporarily shutdown). OSIRIS logs information about process execution on stable storage, and this information is used during recovery to reestablish the normal state of operation. However, there is currently no mechanism in OSIRIS that allows to recover process execution if the node fails permanently. As the node is the only responsible for continuing process execution and storing the state of the process, this kind of fault is never detected and process execution is blocked. The issue we address in this paper is thus how to modify the system so that permanent node failures can be handled, without sacrificing the independence of nodes. Any solution must satisfy two conditions: i) incorporate a monitoring mechanism that detects faults and enacts appropriate corrective actions, and ii) provide means for storing process state in a way that is reliable, independently of specific nodes involved in the computation.

4. SHEPHERD

In this section and in the following one, we describe *Shepherd*, our solution for fault-tolerant decentralized process execution within OSIRIS. The main idea behind Shepherd is to introduce a new layer whose role is to supervise the execution and take appropriate actions in case it gets blocked due to failures. We call this layer the *Shepherding Layer*, and the components belonging to it *shepherds*.

More concretely, the shepherds are responsible for triggering and regularly monitoring the execution of process activities on OSIRIS nodes. If a node involved in the execution of an activity fails, even permanently, they ensure that state of the process is not lost and execution continues.

Regular OSIRIS nodes that are only responsible for the execution of process activities are called *worker nodes*, and correspond to the nodes showed in Figure 1. The architecture of the Shepherding Layer must fit to the decentralized nature of process execution in OSIRIS, and must be itself robust against faults, to guarantee that monitoring activities continue even if a certain number of shepherds crash. For this reason, the execution of a single activity is supervised by a pool of shepherds, one acting as a leader and the others as replicas that can replace the leader if needed. It is also necessary to ensure that, during the execution of an activity, the process state needed to start the activity is persisted in a fault-tolerant way. If a node fails, the process state must be recoverable to start again the computation. Rather than leaving the responsibility to do so to the shepherding layer, we have opted for a more modular architecture, and

introduced also a *Shared Memory Layer*, that acts only as a globally accessible, distributed, fault-tolerant storage. This layer can be seen as an associative array, used to associate process migration tokens to arbitrarily chosen keys. We further require that this data structure exhibits transactional behaviour, in the sense that it guarantees atomicity and consistency of read/write operations. Literature on Distributed Hash Tables (DHT) provides several examples of completely decentralized distributed systems that offer this functionality (e.g. [10, 12, 9, 18]), and in this paper we do not discuss further the precise implementation we have adopted. As it will become clear later in this section, only worker nodes communicate with the Shared Memory Layer. These nodes act as clients, and the layer has a purely passive role with respect to process execution. For this reason, in the examples given in the course of this Section we will consider the layer as an abstract global entity.

It is important to notice that, while logically separated from the process execution layer, the Shepherding and Shared Memory Layers are hosted on the same nodes that participate to a normal OSIRIS cluster. Components responsible for providing their functionality coexist, on each OSIRIS node, with other components. This means that any physical node in an OSIRIS cluster can take the roles of a shepherd and a worker node at the same time.

At the core of Shepherd there is a new process state migration algorithm. Essentially, the algorithm proceeds like this. When an activity must be executed, a pool of shepherds communicates to a worker node an *activation key*, that allows the node to retrieve the migration token from the shared memory. During the execution, the shepherds continuously supervise the worker node. If the node crashes, they pick another one in a way that is explained later in this section. If the activity terminates regularly, the worker will write the migration token for the next activities on shared memory using a new activation key, and the shepherds will receive from the node all the information needed to pass it on. Based on this information, they deliver the new activation key to another pool of shepherds. Notice that shepherds themselves are only indirectly involved in the execution of the process. The logic about how to process migration tokens and how to execute activities is still encapsulated within the OSIRIS processing layer. However, the Shepherding Layer has the responsibility to perform routing within the cluster and late binding of activities to concrete service instances to achieve fault-tolerance. This split of responsibility changes the original process migration algorithm only slightly, as it adds only one level of indirection by introducing a new kind of OSIRIS node (the shepherd) for monitoring, routing and binding.

4.1 The Shepherd Migration Algorithm

Before we proceed, we need to specify a number of assumptions on our communication model, our model of faults, and the Shepherding Layer. Regarding communications and faults, we will assume here partial synchrony, and a fail-noisy model of distributed system [15]. In this abstraction, communication links are perfect links (reliable channels), processes crash with a crash-stop behavior, and an eventually perfect failure detector is available at any node. In other words, if no process at either end of a channel crashes, messages are eventually reliably delivered (not duplicated, not lost) and delivery is acknowledged. Processes that crash do not later join again the execution of the algorithm (notice that this assumption does not forbid that crashed processes join again the system, nodes that crash and recover might become available to participate in a later execution of the algorithm). An eventually perfect failure detector allows nodes to detect failure of other nodes. This detector assumes that processes that do not send a regular heartbeat have crashed. Given the partial synchrony of the system, some heartbeats might not be received on time due to message congestions, and a process might be falsely suspected to have crashed. However, suspicions about processes that start again to deliver heartbeats are revised, and the detector will eventually suspect only those processes which have actually crashed.

Finally, we assume that all services invoked by worker nodes are *fail-safe*. By this we mean that a service that fails does not leave behind any uncompensated side-effect. The case of services that require explicit undo, rollback or compensation in case of failure is outside of the scope of this paper, and might be the suject of future research.

Regarding the Shepherding Layer, we make here the following assumptions:

i) Each worker node in OSIRIS is assigned to a pool of shepherds, and each pool monitors several worker nodes. The set of nodes monitored by a pool of shepherds is called its *herd*. Note that a herd may contain worker nodes with different service types;

ii) Each pool of shepherds has a leader. This leader is the only node with whom a worker node executing an activity communicates, and the only that can take actions (like activating another worker if one crashes, or forwarding the state of the process). If the leader crashes, a single other leader is eventually elected;

iii) Shepherds in a pool share all state related to a currently monitored process activity execution. This knowledge includes the activation key, the identity of the worker node currently executing the activity, and any intermediate piece of information that it communicates to the pool. In order to maintain consistency of this internal state, all communications from worker nodes to the pool have transactional nature, and it is guaranteed that if a message from the worker node to the pool is accepted, the message has reached a quorum of shepherds;

iv) Given an arbitrary service type T, any shepherd is able to deliver messages to the leader of another pool whose herd includes at least one worker node hosting a service instance of type T.

Precise details on how the Shepherding Layer is organized, an in particular the algorithms used for assigning worker nodes to herds, choosing replicas, performing leader election, enabling transactional communications, and routing from a shepherd to another are given in the next section.

4.1.1 Basic Case

Consider again the process definition given in Figure 1. An example of how it would be executed in Shepherd is given in Figure 2. Nodes in figure 2 correspond to those in Figure 1. The nodes in the midldle part of the picture are worker nodes, denoted with their service type (A, B and C). Nodes

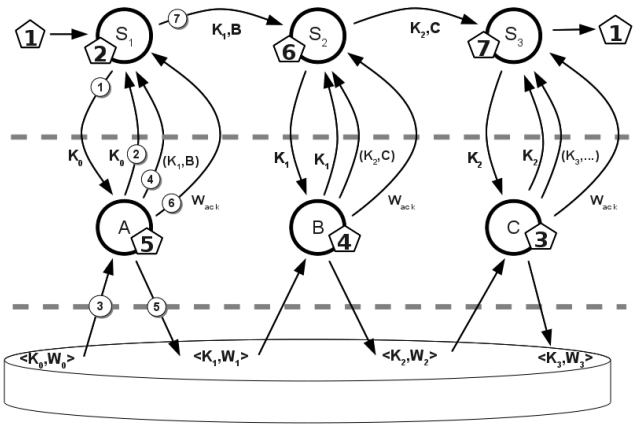

Figure 2: Process Execution with Shepherds

S_1, S_2, S_3 in the upper part of the picture are the leaders of the pools of shepherds monitoring these workers. Note that, though a single worker is shown here for clarity, the herd of each shepherd contains various nodes, with arbitrary types. Finally, the bottom part of the picture shows the Shared Memory Layer.

We assume that another node (e.g, node no. 1, not shown here) starts the process, storing on shared memory an initial whiteboard W_0, and communicates the key K_0, needed to retrieve it, to the shepherd S_1. Let us now consider the steps occurring during the normal execution of this fragment of process instance.

1. `activate(K)` S_1 picks a member of its herd (A), and activates it, communicating to it the key K_0. The exact way the choice is made (e.g., based on load information) does not influence the way the process is executed.

2. `superviseMe(K)` Using the key it received, node A sends back to the shepherd that activated it a message to acknowledge supervision.

3. `get(K)` Using the key it received, node A retrieves the process state from shared memory and proceeds to the execution of its task.

4. `prepareWrite(K,B)` As the task is completed, node A generates a new key K_1 and sends it to its shepherd. Beside the key, the node also communicates to the shepherd the type of the successor node.

5. `put(K,W)` Node A stores the migration token W_1 on shared memory using the key K_1.

6. `acknowledge(W)` Node A communicates to its shepherd with a write acknowledgement that the whiteboard is correctly stored on shared memory. After this communication, the responsibility of node A terminates.

7. `migrate(K,B)` In order to proceed with the execution, the shepherd identifies another shepherd of the appropriate type (in this case B), and sends it the key.

8. `delete(K)` After the key has been delivered, the responsibility of the shepherd S_1 (for this execution step) ends. The shepherd can thus safely delete the value stored on shared memory with key K_0. This step is not shown in the picture to avoid visual cluttering.

This protocol can be iterated over and over each time a shepherd receives an activation key.

4.1.2 Worker Node Failures

Let us now suppose, without loss of generality, that node B fails. Failure must lead to two effects. First, a new service instance B', of the same service type of B, must be activated in order to recover process execution. Second, any side-effects induced by the invocation of B on the shared memory must be undone. As we are assuming fail-safe services, we do not need to consider side-effects outside the shared memory.

If a shepherd suspects, by means of his eventually perfect failure detector, that a worker node has crashed, it will undertake a different action, depending on the stage of the migration algorithm that has been reached.

In case B fails after its activation (step 1) but before having sent the final write acknoledgement (step 6), the shepherd must continue the execution of the process by activating another worker node B', again using key K_1. B' will then save the results of its execution in the shared memory using a newly generated unique key K_2', that will be propagated to the shepherd upon correct termination. B' will be picked from the shepherds herd. If the herd does not contain a node of the appropriate type, S_2 will delegate to another pool af shepherds. If the failure happen after step 6, no action is required, as the responsibility of the worker has already ended.

Regarding side effects on shared memory, if B fails before sending a prepareWrite message (Step 4), then no value has been written on shared memory, and the shepherd does not need to take any further action. Otherwise, the failed execution of B might have produced side effects. It is important to observe that any side effects are completely transparent for B', and do not affect in any way the correct execution of the process, or of any other concurrent process instance. The migration token possibly written by B is only accessible using the proper activation key, which is not passed on by the shepherd. However, side effects should be undone to avoid wasting storage. This can be done by deleting the value associated to the key K_2.

Finally, the shepherd may have decided to activate B' because it erroneously suspected B to have crashed (e.g due to networking malfunctions). In this case, when this false suspicion is revised, the shepherd should nevertheless force B to "crash", to avoid that it waits forever.

4.1.3 Shepherd and Shared Memory Failures

Let us now consider that a shepherd node fails. In particular, we will consider the critical case of failures of the leader. In order to recover from this kind of failures, the only condition that has to be met is that the pool of shepherds always maintains consistent information about the state of the activity it is supervising. This can be achieved by means of distributed transactions. Whenever important information is exchanged between the worker node and the shepherds (e.g., step 2), or between two pools of shepherds (step 6), the communication must be transactional, for instance happen in the framework of a distributed atomic commit algorithm, such as the Paxos commit protocol ([6]). As information about the execution is kept consistent in the pool, a newly promoted leader can always recover the latest state, and

continue the algorithm.

Notice that the presence of a leader is only needed in critical parts of the algorithm, like deciding to replace a failed worker. In the absence of a leader, a worker node can continue to communicate with the pool. However, if it remains isolated from a leader for a bounded period of time, the node will voluntarily abandon the execution of the algorithm (i.e. crash). This is to avoid that a node waits forever in case all shepherds failed. It is also possible that the node actually crashes while waiting for a new shepherd to be elected. This crash is anyway eventually detected by the new leader, and handled by reactivating a new node of the same type.

The Shared Memory Layer has, with respect to the migration algorithm, only a passive role. Worker nodes executing activities just read and write from this layer by means of distributed transactions. The layer itself does not need to take any action in case these operations fail. A worker node can repeat read-write operations multiple times, until they eventually succeed. Finally, our assumption that activation keys are single-use further excludes the possibility that different worker nodes might access the same value and bring the system in an inconsistent state.

5. THE SHEPHERDING LAYER

In this section, we describe the internal organization of the main constituent part of the shepherd fault tolerance mechanism, the Shepherding Layer. In particular, we will describe *a)* How nodes are assigned to the herd of a shepherd, how several shepherds coordinate to form a pool, and how leader election within a pool proceeds; *b)* communication between a worker node and a pool of shepherds; *c)* how shepherds can route process execution to another pool during process instance migration. This routing mechanism essentially provides late binding of process activities, like in the original version of OSIRIS, but with the difference that both routing and load-balancing are based on a purely decentralized P2P design.

5.1 Shepherds and Herds

An OSIRIS cluster is composed of a large amount of computational resources, the worker nodes. In Shepherd, these resources are assigned to special nodes, the shepherds, that are responsible for monitoring them. Every OSIRIS node can play the role of shepherd, but only a limited number of them assume this role. Whether a node becomes shepherd is determined randomly at the startup of the node. How to maintain an optimal number of shepherds in the system is subject to forthcoming performance evaluation. Given the distributed nature of the system, an even assignment of resources should be reached without relying on central components, i.e. in a purely P2P fashion.

These requisites are very close to those underlying the design of Distributed Hash Tables. Indeed, it is possible to re-adapt for the Shepherding Layer the very same concepts found in the routing layers of DHT systems. In Chord [19] and its derivatives, each element in a set of resources, that includes physical nodes and keys, is assigned a unique identifier obtained applying a consistent hashing function. These identifiers can be imagined as evenly spread on what is called an identifier circle. Each node on the identifier circle becomes then responsible for a number of keys. This set is called its *identifier space*. More precisely, the identifier space of a node n consists of all keys such that the first node

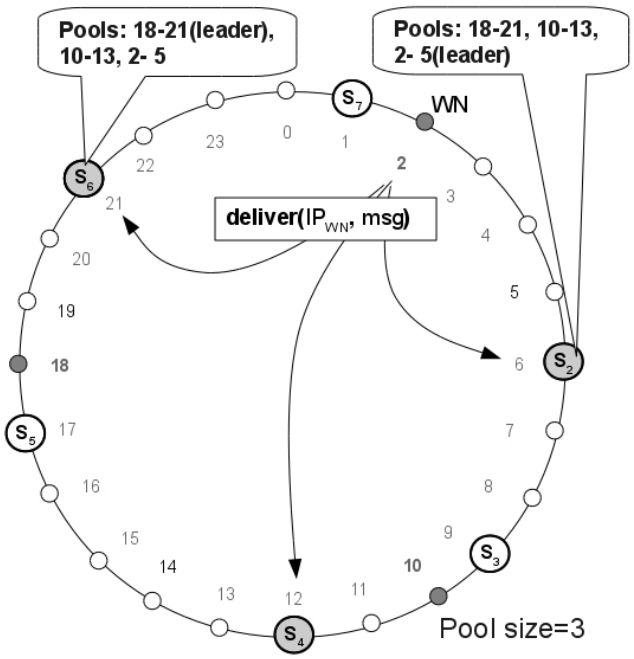

Figure 3: The shepherds ring

following the key in the circle is n. When nodes join or leave the circle, only a small amount of keys has to be relocated (if there are N nodes and K keys, this amount is O(K/N)). Lookup of resources associated to keys relies on an efficient routing algorithm, that guarantees O(log(N)) hops before reaching the node responsible for a resource. Shepherd uses a modified Chord ring to assign responsibilities over worker nodes to shepherds and perform routing. We call it the *shepherds' ring*. All nodes present in the system are assigned an identifier obtained by hashing their IP address. Shepherds become responsible for those worker nodes whose identifiers fall in their identifier space. The routing mechanism from Chord enables worker nodes to efficiently deliver messages to the shepherd which is currently responsible for them. More precisely, instead of offering a `put(Key, Value)` primitive, the shepherds ring offers a `deliver(IP, message)` primitive, that delivers a message to the shepherd responsible for the node with given IP. After entering the shepherds' ring, a worker node can join the herd of the shepherd responsible for it by communicating its identity to the shepherd. This is done by sending a join message containing the IP of the node: `deliver(IP, join(IP))`. After receiving the IP address of the worker node, the Shepherd is able to contact the node, and to monitor it when required. The shepherd initially also communicates its address to the worker nodes, so that communication can flow in both directions. After this link is established, shepherd and worker nodes exchange regularly an heart beat to support detection of faults.

5.2 Shepherd Pools

While each node is assigned to a main responsible, more than one shepherd are actually in charge of monitoring it, acting as a pool. Shepherds in a pool share knowledge about a set of worker nodes. Furthermore, they maintain consistent information during the steps of the migration algorithm

involving those nodes. Pools are chosen using a strategy analogous to that used for the identification of node replicas in DHT systems with symmetric replication schemes ([5]). More precisely, node identifiers are partitioned in a set of congruence-modulo equivalence classes, and a shepherd which is responsible for a given identifier is indirectly responsible for all nodes with identifiers in the same equivalence class. The responsible for node n is also the leader for the pool monitoring n. If the responsible for n leaves the ring (e.g., because it crashes), the first subsequent shepherd in the ring becomes responsible, and joins the pool. It must then retrieve from other members of the pool the information related to the worker it has inherited.

During the execution of the migration algorithm (Figure 2, e.g. step no. 2), a worker node must communicate information to the pool of shepherds that monitors it. The communication must have atomic nature: whenever a node attempts to write a value to the pool, the write will succeed only if a quorum of members of the pool actually recorded that value.

It is immediate to see that this problem is analogous to that of storing a value in DHT systems offering transactional guarantees. Different ways of achieving atomic commit in DHT systems have been proposed [8, 7, 10]. Shepherd adopts a solution similar to that presented in [10], which assumes a symmetric replication scheme and uses a modified version of the Paxos commit protocol [6]. The `deliver(IP, message)` primitive used in the shepherds' ring has transactional properties: whenever a message is delivered to the shepherd responsible for the worker node, the message is guaranteed to have been received also by a quorum of all members of the related pool. The algorithms used for determining pools and performing atomic delivery are trivial modifications of those presented in [5] and [10], and it is not necessary to describe them here in further detail.

The primitive `deliver(IP, message)` is the only one used by worker nodes to communicate with their pool. Even if a leader fails during the migration algorithm, a worker node can continue to communicate values to the pool using that primitive. As a new leader is elected, the leader is guaranteed to read the most up-to-date information about the worker node.

Picture 3 shown an example of shepherds' ring. Each node is associated to an identifier obtained by hashing its IP address. The worker node denoted with WN has identifier 2. The shepherd responsible for WN is the first that follows WN in the ring, in this example S_2. This shepherd is also the leader of the pool monitoring WN (S_2, S_4 and S_6). These shepherds are assigned to the pool of WN because they are responsible for the id that fall in the same congruence modulo class as WN. As the pool size in this example is fixed to 3, this class is $[2, 10, 18]$. At the moment of joining the ring and during the execution of the migration algorithm, the worker node WN can transactionally deliver to the pool a message using the primitive `deliver(`$IP_{WN}, message$`)`.

5.3 Late Binding

OSIRIS nodes host instances of services whose types fall into a given set of service types. Each shepherd is responsible for a number of nodes, each of which might host instances of services of various service types. In the following we say that a shepherd *provides type* T if it monitors instances of type T. Notice that each shepherd provides multiple service

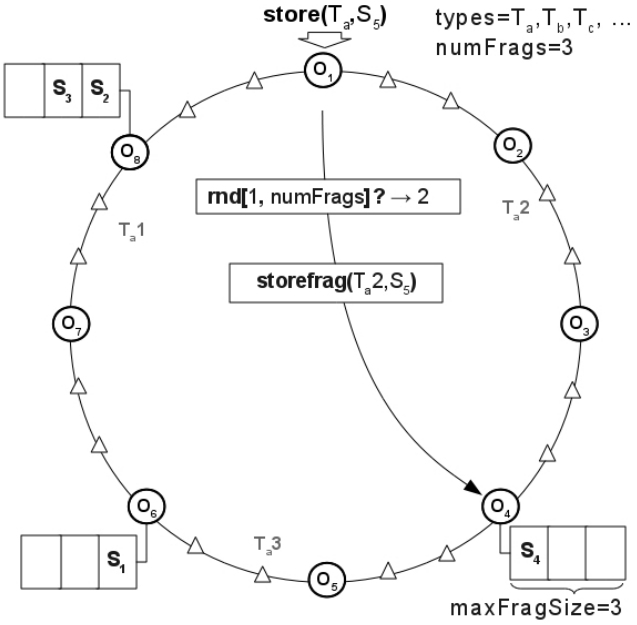

Figure 4: The binding ring

types, and each service type is provided by many shepherds. When a worker node has completed the execution of an activity, the leader of the pool monitoring it must migrate the execution by contacting one or more other shepherds (Figure 2, step no. 7). These shepherds must provide the right service types. After they receive an activation key, they will use it to activate a concrete instance of the right type from their herd (late binding).

Identification of subsequent shepherds cannot be efficiently made based on the shepherd's ring alone. Observe that a potentially large number of nodes might deploy instances of the same service type. A common assumption made in systems like Chord is that resources can be assigned a unique identifier, corresponding to the hashed value of the key with which they are stored. These systems are optimized with the implicit assumption of perfect hashing. Our setting is instead best modeled with a data structure in which a key (the service type) is associated to a list of identifiers (the shepherds) and the basic operation would be, given a key, to pick one element from the list. Notice that, as the size of the system grows, these lists can become very large, and it would thus be infeasible to keep the full list of shepherds as a single value associated to a key.

One way to address this problem is to split the list into fragments with bounded size, and assign each fragment to a different node. This solution is illustrated in Figure 4. In the picture, a list containing shepherds providing type T is stored as a serie of fragments distributed over 3 different nodes. It is important to stress that the *binding ring* shown in Figure 4 is logically separated from the shepherds' ring illustrated in the previous section. Nodes on the binding ring $(O_1, .., O_8)$ are generic OSIRIS nodes, connected through a Chord ring. Nodes are responsible for resources that correspond to list fragment identifiers rather than node IPs. For ease of exposition, in the picture we have chosen to show a few keys rather than the corresponding identifiers. Also note that, although the picture shows only the list for type

Algorithm 1 store(T, V)

Require: a key T and a value V to store in a T related list
  $numFrags = get(T)$
  if $!numFrags$ **then**
    $put(numFrags, 1)$
    $storeInFragment(T1, V, 1)$
  else
    $f = rnd[1..numFrags]$
    $storeInFragment(T, f, V, numFrags)$
  end if

Algorithm 2 storeInFragment($T, f, L_v, numFrags$)

Require: a fragment key T, a fragment index f, a value V to store in a list fragment, and the number of fragments currently present for the list.
  $T_f = concat(T, f)$
  if !this.responsible(T_f) **then**
    $successor = lookup(T_f)$
    $successor.storeInFragment(T, numFrags + 1, newFragment, numFrags + 1)$
  else
    $fragment = this.get(T_f)$
    if $|fragment| == maxFragSize$ **then**
      $put(T, numFrags + 1)$
      while $|fragment| \geq I/2$ **do**
        $values.add(fragment.first)$
        $fragment.remove(fragment.first)$
      end while
      $T_{new} = concat(T, numFrags + 1)$
      $successor = lookup(T_{new})$
      $successor.storeInFragment(T, numFrags + 1, values, numFrags + 1)$
    end if
    for all $VinL_v$ **do**
      $fragment.add(V)$
    end for
  end if

T_a, each OSIRIS node will in general store multiple list fragments, associated with different service types. How are the list of shepherds constructed? Algorithm 1 provides an operation `store(T,S)` to store a shepherd address S in the list associated to the service type T. The algorithm will initially query the ring (through the usual *get* operation) for the key T, and retrieve the count $numFrags$ of the fragments composing the list associated to this key. It will then randomly pick an integer value x in the interval $[1..numFrags]$ and store the shepherd in the fragment identified by the key T_x using the procedure `storefrag(T,S)` showed in Algorithm 2. If the size of the fragment does not exceed a predefined maximum value $maxFragSize$, the shepherd S will be saved in that fragment. If instead adding the shepherd would violate the maximum allowed size, the fragment is first split in two new fragments. More precisely, before storing the new value, the node responsible for T_x will transfer half of the content of its fragment to a new fragment with key $T(numFrags + 1)$, and increment the count of fragments for the list.

Once a worker node has joined the herd of a shepherd, it can publish to it the set of service types for the instances it hosts. The shepherd will then publish itself on the binding ring, based on the set of service types it provides.

Through this data structure it is immediate, given a service type, to retrieve a shepherd providing that type. The simplest possible algorithm selects a random fragment and then returns a random shepherd from that fragment.

5.4 Load Balancing

Instances of the same type can be considered completely equivalent from a semantic point of view. This means that performing a random binding as explained above is sufficient to guarantee the correctness of the routing, and enable late-binding. However, binding must take into consideration other factors that are crucial for the performance of the overall process execution, in particular the load of nodes. Once a service node has joined the shepherds' ring, it can periodically notify its shepherd about its load status. A shepherd, in turn, can communicate to the nodes responsible for the fragments in which it lies aggregated load information, to be used for shepherd selection. In the binding ring, values stored in fragments of the list associated to a type T are of the form $< S, L >$, where S is a shepherd providing T, and L is a value denoting their current average load level. This is not shown in Figure 4 for simplicity.

In order to pick the least loaded shepherd providing a given type, we can store a reference to the least-loaded shepherd and keep it dynamically up to date. For a given type T, we will store the reference to the least loaded shepherd providing T with the special key T_{best}. The value stored at T_{best} is a tuple $< S_{best}, Load_{best} >$, indicating the identity of the shepherd and its load level.

Suppose now the list for type T is split in f fragments assigned to nodes $O_1, .., O_f$. Whenever O_i receives an update from a shepherd in its fragment, it checks if the least loaded node within the fragment has changed. Suppose this is the case, and S is now the least loaded shepherd within the fragment. O_i will then candidate S to become the new S_{best}, by triggering a *contest*. This is done by propagating through all its successors in the fragment list $O_i, O_{(i+1)}, \ldots$, a packet containing information about the candidate S and its load. Each node will substitute the candidate with its best shepherd, if this is better then the candidate, and forward the packet. The process ends when the packet reaches again O_i, which will then replace T_{best} with the candidate that won the contest.

Observe that, if no value for the best shepherd has been published yet, a random shepherd obtained with the algorithm given above can still be used. Furthermore, while very accurate estimations for the load are beneficial for the performance of the system, a lower level of freshness for this values can be tolerated, as it does not affect the functionality of the system.

6. EVALUATION

In this section, we compare the Shepherd approach to fault-tolerance with the original OSIRIS approach, identifying benefits and possible drawbacks of our solution, and we provide a preliminary analysis of the performance of the system. It has to be remarked that this paper describes ongoing research, and no complete implementation of Shepherd is currently available. For this reason, the evaluation carried out in this section is only qualitative. Quantitative empirical performance measurements shall be carried out in the future. The table 6 tries to summarize the differences

	OSIRIS	Shepherd
Execution Style	P2P	P2P
Monitoring	N/A	shepherds pool
Activity Execution	1 worker node	1 worker (>1 in case of failures)
Routing/Late Binding	worker node	shepherd
Failure Model	crash-recovery	crash-stop
Failures	only temporary	all
Failure detection	global timeout	heartbeats
State Checkpointing	on local stable storage	on transactional DHT
Overhead during execution	none	monitoring
Migration	2PC	Paxos Commit
Metadata management	centralized, pubsub	fully decentralized (DHT)

Table 1: Comparison between OSIRIS and Shepherd

between the two architectures at a glance.

Shepherd addresses a number of issues related to how the original OSIRIS treats faults. In OSIRIS, only temporary faults can be fully managed. OSIRIS follows a crash-recovery model for failures [15]. If a node that is currently involved in the execution of a given activity crashes, execution of the related process is blocked until the given node recovers. During execution, a node keeps on local stable-storage enough information to reconstruct the state of execution at the moment of the crash. In particular, a node is able to identify whether it had already invoked a service and received a response from it. This guarantees that a node which has received a migration token will eventually complete its activity, unless a permanent failure occurs. This makes simpler to guarantee *exactly once* invocation of underlying services. However, the solution provided in OSIRIS does not address the case of processes that crash and do not recover. If a node crashes permanently, execution is blocked forever.

This solution is acceptable in computational clusters which run under direct human supervision, and where hardware failures are unlikely events, which can be repaired in a relatively short time, and do not normally lead to data loss or corruption. However, they are not appropriate to very-large data center that employ inexpensive commodity hardware, in which crashes are very frequent events, and often data cannot be recovered. As failures become more frequent, the waiting times needed to continue execution in case of temporary failures, i.e. until a machine is repaired and manually rebooted, become unacceptable.

Shepherd takes a pessimistic point of view, treating hardware and in particular storage as essentially unreliable. This point of view is intrinsic in the assumption of a crash-stop model for failures. The system is able to proceed with execution independently of crashes of any specific machine, up to a reasonable amount of failures. Of course, there is a price to pay connected to this higher resilience to faults and higher level of automation. The system is more complex, and involves more coupling between different nodes. A performance penalty, in terms of number of messages exchanged, is associated with monitoring and saving state redundantly in a distributed way. An exact quantification of this overhead is not possible without an empirical evaluation.

Another main difference between Shepherd and the original OSIRIS is the adoption, in Shepherd, of a completely decentralized architecture. Many components of the architecture of OSIRIS, and in particular those related to the handling of metadata, are centralized. In [17], the authors hint to the fact that these components could be decentralized. However, no concrete solution is proposed. The architecture proposed here for Shepherd is completely decentralized, and based on Chord-like routing.

7. CONCLUSION AND FUTURE WORK

This paper presents ongoing work on Shepherd, a novel fault-tolerance mechanism for the distributed process execution engine OSIRIS. Shepherd includes two new logical layers, a distributed monitoring overlay, responsible for detecting process execution faults and taking appropriate remedial actions, and a distributed, fault tolerant transactional object storage, responsible for temporarily storing intermediate process execution state. At the core of the approach there is a fault-tolerant process state migration algorithm. We have shown that the Shepherd approach enables to treat a wider class of faults than the original OSIRIS architecture, including permanent failures of nodes currently executing activities. Beside this advantage, the shepherd architecture introduces a decentralization of all OSIRIS meta-data repositories, removing the last traces of centralization in OSIRIS and thus potential bottlenecks and single points of failure.

The main contributions of this paper are the description of the overall architecture of Shepherd, of the migration algorithm, and of the monitoring layer, that is based on a transactional DHT overlay [9], but extends it with additional features needed to perform service late binding.

Future work will focus on performing an experimental evaluation of Shepherd, particularly to determine the efficiency of the approach in real-world cases. Additional interesting topics have emerged during the development of Shepherd. In particular, tailoring of the protocols used for distributed transaction to our specific needs would enable us to reduce the communication overhead of the migration algorithm. Introducing locking schemes for the Shared Memory Layer might also be beneficial. Finally, fault-tolerance policies might be made customizable by the user with the introduction of an economical cost model allowing for trade-offs between consistency and performance.

8. REFERENCES

[1] Boualem Benatallah, Marlon Dumas, and Quan Z. Sheng. Facilitating the rapid development and scalable

orchestration of composite web services. *Distrib. Parallel Databases*, 2005.

[2] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[3] Erik Elmroth, Francisco Hernandez, and Johan Tordsson. A Light-Weight Grid Workflow Execution Engine Enabling Client and Middleware Independence. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 754–761. Springer Berlin, Heidelberg, 2008.

[4] Wolfgang Emmerich, Ben Butchart, Liang Chen, Bruno Wassermann, and Sarah Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3:283–304, 2005.

[5] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *DBISP2P'05/06: Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, 2005.

[6] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[7] Boris Mejías, Mikael Högqvist, and Peter Van Roy. Visualizing Transactional Algorithms for DHTs. In *P2P '08: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, pages 79–80, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Boris Mejías and Peter Van Roy. The relaxed-ring: A fault-tolerant topology for structured overlay networks, 2008.

[9] Boris Mejías and Peter Van Roy. Beernet: Building Self-Managing Decentralized Systems with Replicated Transactional Storage. *In International Journal of Adaptivee, Resilient and Autonomc Systems (IJARAS)*, 1:1–24, 2010.

[10] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In *Proc. of the CoreGRID Symposium CoreGRID*, 2007.

[11] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. From centralized workflow specification to distributed workflowexecution. *J. Intell. Inf. Syst.*, 10(2):159–184, 1998.

[12] Athicha Muthitacharoen, Athicha Muthitacharoen, Seth Gilbert, Seth Gilbert, Robert Morris, and Robert Morris. Etna: a fault-tolerant algorithm for atomic mutable dht data. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2005.

[13] Stefan Pleisch and André Schiper. Fault-tolerant mobile agent execution. *IEEE Trans. Comput.*, 52(2):209–222, 2003.

[14] Wenyu Qu, Hong Shen, and Xavier Defago. A survey of mobile agent-based fault-tolerant technology. *Parallel and Distributed Computing Applications and Technologies, International Conference on*, 0:446–450, 2005.

[15] Luís Rodrigues Rachid Guerraoui. *Introduction to to Reliable Distributed Programming*. Springer Publishers, Berlin, Germany, 2006.

[16] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-J. Schek. Peer-to-Peer Process Execution with Osiris. In *Proceedings of the 1 st International Conference on Service-Oriented Computing*, 2003.

[17] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-J. Schek. Scalable peer-to-peer process management - The OSIRIS approach. In *Proceedings of the 2 nd International Conference on Web Services (ICWS'2004)*, pages 26–34. IEEE Computer Society, 2004.

[18] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-based consistency and availability in structured overlay networks. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[19] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[20] Xinfeng Ye. Towards a Reliable Distributed Web Service Execution Engine. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 595–602, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Ustun Yildiz and Claude Godart. Synchronization Solutions for Decentralized Service Orchestrations. In *ICIW '07: Proceedings of the Second International Conference on Internet and Web Applications and Services*, page 39, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Weihai Yu. Decentralized Orchestration of BPEL Processes with Execution Consistency. In *APWeb/WAIM '09: Proceedings of the Joint International Conferences on Advances in Data and Web Management*, pages 665–670, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Weihai Yu. Scalable Services Orchestration with Continuation-Passing Messaging. In *INTENSIVE '09: Proceedings of the 2009 First International Conference on Intensive Applications and Services*, pages 59–64, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Weihai Yu and Jie Yang. Continuation-Passing Enactment of Distributed Recoverable Workflows. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 475–481, New York, NY, USA, 2007. ACM.