Distributed Snapshot algorithm for multi-active object-based applications

Michel J. de Souza[1], **Françoise Baude**[2]

[1]Departamento de Ciência da Computação
Universidade Federal da Bahia (UFBA) – Salvador, BA – Brasil

[2]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

michel@dcc.ufba.br, Francoise.Baude@unice.fr

Abstract. *This paper exposes an adaptation of the classic algorithm for consistent snapshot in distributed systems with asynchronous processes due to Chandy&Lamport. A snapshot in this context is described as the consistent set of states of all involved communicating processes that allows recovering the whole system after a crash. The reconstructed system state is consistent, even if messages injected into the system from the outside while the snapshot was ongoing may have been lost (if such messages can not be replayed). We expose how to adapt this algorithm to a particular distributed programming model, the Active Object model (in its multi active version). We applied it successfully to a non trivial distributed application programmed using Active Objects serving as a publish/subscribe and storage of events middleware, dubbed the EventCloud.*

1. Introduction

When a system is dealing with a huge amount of data, it is important to have a backup mechanism that allows its recovery in case of disaster, loss of energy, etc, which in the sequel, we assume requires recovering the whole system, not only one or a few processes. In practice, to accomplish this task a snapshot algorithm can be used to regularily take a consistent and global, even if distributed, checkpoint. A snapshot is indeed the process of collection of data that allows to create a global consistent image of the system state. This paper proposes a snapshot algorithm that works as an adapted version of the classic Chandy-Lamport algorithm [Chandy and Lamport 1985]. The algorithm is adapted in order to suit the constraints from the use of the Active Object programming model that is used to program our distributed system dubbed the EventCloud. Its aim is to act as a storage of events, and allow end-users to retrieve them in an asynchronous manner thanks to a publish-subscribe model. The system is architectured along a structured peer-to-peer overlay network, and aims to be deployed on distributed physical infrastructures (private cluster, private/public cloud, etc.).

Snapshot algorithms assume initial input data is available and could be used ultimately to recover the execution from the very beginning if needed. On the contrary, in our case, input data is coming continuously from the outside, in the form of publications and subscriptions injected inside the EventCloud. It is not realistic to assume we can log all this exterrnal information (as the log would replicate the data that is eventually stored in the EventCloud). Consequently our solution can suffer of publications loss in some circumstances; however, we propose an ad-hoc solution for preventing subscriptions loss.

This paper is organized as follows: the section 2 provides the background and some important concepts to understand the work proposed; the section 3 presents the solution in details and how to address the specificities due to the Active Object programming model as used to implement the EventCloud; the section 4 presents the way an EventCloud gets reconstructed out of a taken snapshot, and consequently which are the failure supported assumptions. The last section concludes and summarizes the improvements left as future works.

2. Background

The proposed snapshot algorithm is implemented within the EventCloud software using the ProActive middleware , thus offering both large-scale deployment capabilities of Active Objects on distributed infrastructures and remote communication features between these objects. This section describes the needed concepts.

2.1. Proactive

ProActive [INRIA and UNS 2000] is an open source Java library aiming to simplify the programming of multithreaded, parallel and distributed applications for clusters, grids, clouds, and any distributed infrastructure in general.

Some ProActive features useful in the sequel are the following. Activities that can execute in parallel take the form of distributed, remotely accessible objects. An active object is an entity with its own configurable activity. Each active object has its own thread of control (so the term "active") that is in charge of selecting and serving the next method call, among those awaiting to be served. Requests to execute method calls are automatically stored in a queue of pending requests. Requests are sent in a FIFO manner on the communication channel connecting a caller and a callee. This FIFO behaviour happens because of a Rendez-Vous protocol: an acknowledgement from the callee is sent back to the caller, to commit the request is stored in the request queue waiting to be served in an asynchronous manner. Consequently, the caller can not initiate a second request before receiving this acknowledgment, ensuring FIFO placement of corresponding requests in the queue in case they target the same callee. By default, the serving policy of requests stored in a queue is FIFO, and one request is served by the control thread at a time. Eventually, active objects interact by asynchronous method calls. Results of method calls are called futures and are first class entities. Callers can transparently wait for results using a mechanism called wait-by-necessity [Caromel 1993]. A callee transparently updates the corresponding future of a method invocation, which automatically wakes up the awaiting caller if it is already blocked waiting for the result.

A distributed or concurrent application that uses ProActive is composed of several active objects. A recent ProActive version [Henrio et al. 2013] replaces active objects by multi-active objects, allowing to serve more than one request at a time. To accomplish this task, one can declare compatibility groups of methods. Requests marked as compatible are allowed to be served in parallel. The algorithm proposed in this work is implemented using the multi-active objects version.

2.2. EventCloud

As described in [Baude et al. 2011], the EventCloud is a distributed store of data following the W3C RDF specification, that handles SPARQL queries across the distributed

architecture. RDF is a fundamental part of the Semantic Web stack offering a graph structured data model where each relation is made by default of a triple (subject, predicate, object). Although RDF uses triples, the EventCloud works with quadruples - an extended version of a triple that adds a fourth element (usually named context or graph value) to indicate or to identify the data source. Events stored in the EventCloud module can be simple or compound. A compound event is made of a set of quadruples with each quadruple sharing the same graph value. This graph value is assumed to be an identifier that uniquely identifies a compound event.

The EventCloud is a module deployed over a CAN [Ratnasamy et al. 2001] structured peer-to-peer network (a D-dimensional space torus topology). Each peer is a multi-active object that can receive requests that are put into a request queue. All the non compatible requests (e.g. adding a peer reference as neighbour request, and routing a request towards are served in a FIFO (First-in, First-out) manner, whereas declared compatible ones can be served in parallel by the multi-active object (e.g. several requests to route a message whose purpose is relative to a publish or subscribe operation in a dataspace zone the peer is not responsible of can be handled in parallel: their treatment consists to forward the message further in the CAN, by sending it to the right neighbour). Each peer has also a handful of Jena datastores [Jena 2014] to store quadruples, intermediate query results, queries, etc. In the sequel, we will simplify by considering only one single datastore is attached to each peer.

The EventCloud works with the publisher-subscriber paradigm where any external user connected to the EventCloud through a proxy can subscribe to receive data that he is interested in, or can publish data. To make this possible, the EventCloud uses the SPARQL query language to formulate the subscriptions.

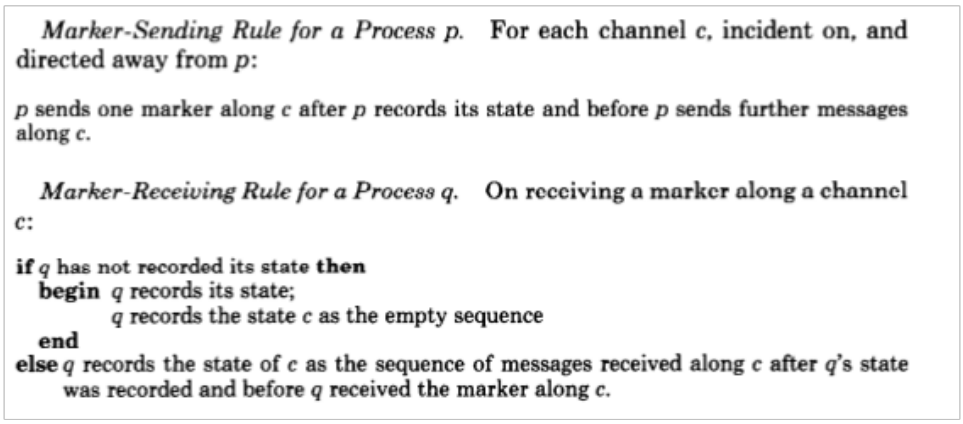
The EventCloud module integrates within the PLAY Platform described in [Roland Stühmer et al. 2011]. The PLAY Platform aims at creating an Event Marketplace at the Web scale. A marketplace works as a search engine dedicated to events and provides visibility to distributed sources of events.

2.3. Chandy-Lamport snapshot algorithm

Before starting to explain how the snapshot algorithm of [Chandy and Lamport 1985] works, it is important to establish some concepts used by it:

- A distributed system consists of a finite set of processes with finite set of channels. It is described by a directed, labeled graph in which the vertices represent processes and the edges represent channels.
- A channel is the connection between two processes by which messages can be sent. It is assumed to have infinite buffers, to be error-free and to deliver messages in the order sent.
- The state of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.
- A process is defined by a set of states: an initial one and a set of actions. An action a in a process p is an atomic action that may change p's state itself and the state from at most one channel c incident on p.
- The communication in a distributed system happens by the exchange of messages. A process only records its own state and the messages it sends and receives.

- To determine a global state, a process must count on the cooperation of the others processes composing the system. They must record their own local states and send this data to a chosen process charged of recording the collected data.
- Unless relying upon a global common clock, the processes cannot record their local states precisely at the same instant.
- The global-state-taking algorithm (snapshot) must run concurrently with the underlying computation not altering it.
- A global state for a distributed system is a set of process and channel states. The initial global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence.



Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:

p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:

if q has not recorded its state **then**
  begin q records its state;
    q records the state c as the empty sequence
  end
else q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

Figure 1. Key rules of the Chandy-Lamport snapshot algorithm [Chandy and Lamport 1985]

The algorithm uses two basic procedures to control the global state construction process and can be initiated for one or more processes. The process in charge of initiating the snapshot algorithm taking records its state spontaneously without receiving a token from others processes. The procedure that represents the sending of a token is named *Marker-Sending Rule*. In this procedure, a process will initially record its state and then will send a token along each channel c directed away from itself. The procedure that represents the receiving of a token is named *Marker-Receiving Rule*. On receiving a token for the first time, a process will record its state (locally). Once a process receives the remaining tokens (from its neighbors), it needs to record the channel state by which the token has arrived. This state is composed by the messages that has arrived after the process has recorded its state and before the receiving of the token. Once the data is collected each process sends it to the process that has started the algorithm. The Figure 1 depicts the algorithm.

3. Snapshot algorithm for the EventCloud system based on Active Objects

3.1. Snapshot Manager

The Snapshot Manager regularly triggers the execution of the snapshot algorithm. If asked, it can initiate an EventCloud instance reconstruction by using the data collected, so it is working as a safe storage place receiving all the peer's data, etc. Reconstruction will be addressed in the next section.

The Snapshot Manager is a multi active object having a reference to an EventCloud Registry. The EventCloud Registry holds reference to EventCloud instances. In this way, whenever the Snapshot Manager needs to perform any operation (start a snapshot, restoration, etc) for one specific Event Cloud, it is going to contact the registry in order to obtain the reference that will allow to fulfill the task.

3.2. Snapshot algorithm adapted from Chandy-Lamport solution

As within Chandy-Lamport algorithm, in the adapted solution presented in this work, channels are assumed to be uni-directional, but between any two peers, a channel in both directions does exist. Moreover, there is a channel from a peer to itself, as a peer can send requests to itself. Differently from original version, each peer collects its data and sends it to the safe storage place; whereas in Chandy-lamport algorithm, the data collected is sent to the peer that has started the algorithm.

Peer data to be saved with the algorithm are as follows:

1. Jena datastore - application data.
2. Request queue relevant information.
3. Topology information: neighborhood table, zone, etc.
4. Some other EventCloud management parameters.

The Figure 2 above depicts our adapted snapshot taking algorithm.

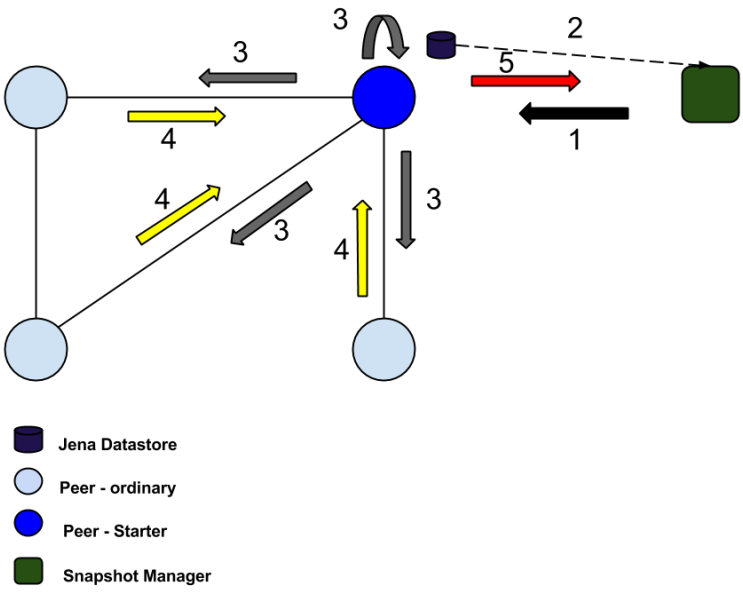


Figure 2. Snapshot algorithm execution process. Depicted from only the starter peer viewpoint.

The **black arrow** (1) illustrates the moment when the Snapshot Manager initiates the snapshot process choosing randomly a peer to trigger the algorithm.

Once the chosen peer to trigger the algorithm receives the request, it records part

of its state by sending its Jena datastore[1] to the safe storage place (Snapshot Manager) - this step is illustrated by the **dashed black arrow** (2). At this point, the peer locally copies part of the data it needs to collect - topology information (neighborhood table, zone, etc), in order to add this data to its corresponding saved state later on (see below).

Once this same peer has performed the tasks described above, it propagates the token towards its neighbors (including to itself). This step is described by the **gray arrows** (3). This ends up the operation named *Marker sending rule* of the Chandy Lamport algorithm (see Figure 1).

On the picture, a **yellow arrow** (4) describes a sending of a token from all the starter peer neighbors to it. This represents only the starter peer perspective. However, on receiving a token for the fist time, a peer will send a token to each one of its neighbors. Notice the figure only depicts the initiator peer algorithm execution, but it runs almost simarily on each peer (except regarding the black arrow).

Regarding our adaptation of the *Marker receiving rule* of the algorithm, the following is taken in consideration. The **grey** (3) or **yellow arrows** (4), the receiving of a token, trigger a key action: ”closing the channel” they are sent onto.

Every peer, including the initiator one, will have to have served n+1 tokens in total (token carried in the form of a remote method invocation), to terminate the execution of the algorithm instance. One token from each of its n neighbors, and another one that it has sent to itself (as each peer might trigger a method call, i.e. send a request, to itself). Indeed, in the Active Object model, the active object can invoke methods on itself; in the EventCloud implementation, this is extensively used. This self-sent token closes a channel from it to itself. In general, each token served by a peer closes the channel directed from the corresponding sender peer to it. Notice that because each peer is an active object, it stores the received requests (messages) in its request queue. Consequently, we have to adapt the Chandy Lamport algorithm to the fact that channels are not easily identified: they are all mixed in each active object single request queue, and are materialized only at the receiving edge of the channel (i.e. if peer A can send a method call MC to peer B because A knows B’ object reference, the unidirectional communication channel that connects A to B gets materialized by the fact that the message MC is stored in the request queue of peer B. Before MC gets served, one can consider that MC is still on the wire between A and B. This is why MC might need to be part of the global snapshot, besides the states recorded at the moment A and B respectively and independently serve their first token).Consequently, the operation to save in transit messages then close a channel once a token from a neighbour has arrived, has been carefully designed, as detailed just below.

One really important information of the global snapshot to be saved is the relevant (subpart of) request queue information of each peer; so, in background, always before the scheduler in charge of selecting which next request(s) can be served takes out a request from the peer queue, an evaluation process happens to decide if this request (the corresponding message) will be locally copied with all the other requests that represents the in-transit messages (these requests will be sent to the Snapshot Manager once the peer has

[1]Jsch [jsc 2014] is a Java implementation of SSH2. It allows to connect to a server and uses port forwarding , X11 forwarding, file transfer, etc. As Jsch doesn’t allow folder transmission, to be able to transfer the Jena datastore it’s necessary to zip its folders.

finished its snapshot taking process). For a request to be saved, three properties need to be verified: the request cannot be a token because a token represents only a control request part of the algorithm managing what needs to be saved; the peer that has received the request needs to be in RUNNING state what implies the peer being active from the snapshot taking algorithm viewpoint and indicates it hasn't yet finished it's data collection of the current snapshot algorithm instance; at last, the directed channel from the peer sender to peer receiver needs to be still open. The reception of a token on a channel closes this channel. Hence, if a peer has sent a token to another peer and after generates new requests to this same peer they won't be saved at receiver side because they have arrived after the token that closed the channel (a strong property of the active object model is that communication between any two objects is FIFO). On the contrary, if a peer has sent a token to a neighbour, but has not received a token from it, any received request originating from this neighbour is to be considered in transit. So it is going to be part of the corresponding channel state in the global snapshot.

At last, having received all the needed tokens, the peer will send the data it locally collected to the Snapshot Manager, thus enriching the peer state that was only partly recorded so far. This data is composed by the relevant request queue information saved during the moment at which the peer gets engaged in the algorithm snapshot (it has received the first token) and the moment it finishes (it has received all the tokens) and the topology information copied previously. This step is described by the **red arrow** (5).

The picture above depicts the snapshot algorithm from only one peer perspective. However, after the other peers have received the token, they will execute the same set of actions (symmetrically): save its state, send a token to the neighbors, etc. The main difference lies in the snapshot algorithm initiation: the Snapshot Manager contacts only the starter peer (the one that triggers the whole algorithm process). After receiving the token each starter peer neighbor will initiate the algorithm. The algorithm ends when each peer has received all the tokens from its neighbors and has sent the data collected (the peer state, neighborhood table, etc) to the Snapshot Manager. The Snapshot Manager is able to detect when a snapshot taking is terminated, because, for each EventCloud instance, the number of peers part of the distributed support is known (it is stored as information in the EventCloud registry). Until it has not received the expected number of peer states, this snapshot instance is not valid to be used for a recovery.

3.2.1. Intuition about the corrrectness of the algorithm

To gain an intuitive understanding of some points into the algorithm, let us devise a scenario with an Event Cloud having 3 peers that exchanged requests that may not all be served when the snapshot taking starts. Two of them are faster than the remaining one, that is, serve their requests in a faster manner: they have already exchanged tokens between themselves while the third peer hasn't treated yet its first token. Each of these two faster peers has already saved its state locally (Jena datastore, zone, neighborhood table, etc) during the first token service; now each proceeds serving requests which may generate new requests sent to the slower peer that hasn't already saved its state. In the meantime, received requests sent from the slower peer gets ready to be served by the faster peers, and they will be tagged as in-transit.

More globally, the question we raise is dictated by the fact that in our setting, snapshot taking happens concurrently with the application execution: in a snapshot (set of local peer states or channel states), can it happen that it contains a given information that is not a consequence of the recorded global state ? If yes, this means the system could be recovered from a snapshot that is not consistent, in the sense that it reflects a situation that the system would never be able to reach if restarted from an older snapshot. It is well-known that in a distributed system, message reception suffers from non-determinism of arrival, this explains why different executions and so, different peer states could arise from a same global snapshot restart. Also, these different executions may not exhibiting the same requests, nor the same request contents. Because our snapshot comprises copy of requests tagged in-transit, we must ensure that it is correct to serve them in the recovered execution. Also, because the snapshot comprises peer local state copies, we must ensure that in these states, only information that is the result of what the snapshot encompasses is present: otherwise, this means the state encompasses causal effect of actions but these actions are not supposed to have arisen as summarized by the snapshot content.

In the original Chandy-Lamport algorithm, there is no need to raise this question: a process engaged in the algorithm is applying rules depicted in Figure 1, but in the meantime, it is not processing any of the messages received on its incoming channels, meaning the state that is distributively recorded in the snapshot is not evolving while it's built. On the contrary, in the Active Object model, a peer must proceed serving the requests (in FIFO order by default) to have a chance to process tokens. That means the numerous tokens related actions like local state and channels state savings can happen while peer state has been affected by applicative-level requests service that must eventually not be "visible" (directly or indirectly) in the snapshot. So, for the algorithm to be correct, it must ensure that these effects are left aside and are not wrongly recorded in the snapshot.

First, peer local state saving happens at the very first token service. It only reflects request service effects that must be part of the snapshot. In particular, none of the served requests yielding to the recorded state was tagged as an in-transit request. Indeed by definition, in-transit requests stored in the queue are positioned after the very first token in the request queue, so always served after the peer local state saving, thus having no impact on the saved local state. Also, no request served before the first token service could correspond to a request that should not be part of the global state because was created by a sender after it recorded its own local state (as communication channels in ProActive are FIFO).

The second case discussion is about channel states that are logged, that correspond to in-transit requests within the snapshot. Could it happen that an in-transit request gets recorded in the snapshot alas it should not have been because its originating action is not part itself of the snapshot ? The risk being that in a replay of the system, the execution takes place in a way such that the request that triggered the one tagged in transit and part of the recovered state does not arise. The answer is no: No in-transit request can be a consequence of a request service whose effect is not part of the snapshot.

Thirdly, while the snapshot taking is running on the distributed system, active objects do not stop serving requests, some being detected in-transit requests, some being received before (resp. after) locally entering the snapshot taking execution (so before (resp. after) the first token service). Except for those received before first token service,

no impact from their service happens on the snapshot recorded state: local state was copied while serving the first token, which atomically propagated tokens on outgoing channels that, once served will close channels. Consequently, any new request sent on these channels (assumed to be FIFO) cannot affect the receiver recorded local state nor corresponding channel state.

3.3. Current limitations

If peers are removed or added to an EventCloud intentionally, according to its associated load balancing strategy, removal or addition of peers is an information that can easily be tracked for recording the effective number of peers. However, notice that supporting removal or addition of peers while a snapshot algorithm is running is left as future work (see section 5). Say another way, the algorithm as it is presented here does assume the number of peers is static, in the sense the topology of the CAN architecture is not changing while the snapshot algorithm is running.

In the current implementation, only one instance of the snapshot algorithm can be running at a time. To initiate the execution of new snapshot algorithm instance the termination of its execution must have been detected. This is a limitation because we may want to periodically put in execution a new algorithm instance, that is, after a certain amount of time a new instance would automatically start, getting a certain overlapping of many global different states. As recovery is triggered from the most recent completed snapshot, it is obvious the less time elapsed since its completion, the best it is as only ”few” work is going to be lost and redone. In fact, to support many instances of the snapshot algorithm run in parallel would simply require to extend the current implementation by making all exchanged tokens and associated treatments triggered by the snapshot algorithm tagged by an instance number. For instance, a channel may be closed by a token service of one specific instance, while it may still be open regarding another instance of the snapshot algorithm execution.

4. EventCloud recovery

Given the snapshot algorithm has allowed to record all the data needed, our next aim is to support the inverse process, that is, reconstruct the system using this information.

4.1. Restoration process

When one decides to reconstruct an EventCloud, the Snapshot Manager will trigger the restoration process. According to the snapshot identification, the Snapshot Manager will access the file that contains management information about the EventCloud in question and the state set (from each peer) that contains all the data: neighborhood table, peer request queue, Jena datastore, etc.

The reconstruction algorithm starts similarly to the process of an EventCloud instantiation: when an EventCloud is instantiated, the peers are created with random identifiers, they do the join process according to the CAN protocol to establish the zone they are responsible for and the neighborhood table, etc. However, in the reconstruction process, the peers will be created with well known identifiers, passed as arguments, as this information is already established. Another difference lies in the join phase that is discarded. This phase is useless when reconstructing an EventCloud because the topology information (neighborhood, zone, zone split history) from each peer is already known.

Once the peers created, the data collected must be sent to the peers and some adjustments must be also done. First, the Snapshot Manager sends the corresponding Jena datastore, by using Jsch library, to each peer. This process will create the needed local connection with Jena datastore. Then, the topology data is sent. The topology data is composed by the neighborhood table, peer zone and the split history; however the active object's remote addresses (handled within a stub object) entries in the neighborhood table from each peer is not valid because all the peers were recreated. So, the stub entries need to be updated. To fulfill this task, each peer sends a 2-tuple (id,stub) to its neighbors. Receiving this 2-tuple, the peer will replace the stub in the neighborhood table that corresponds to the id, concluding the update. The list of 2-tuples (id, stub) was created in the peer creation phase. As described before in this section, the join phase in the reconstruction process is useless because the zone and split history are passed to the peers (they were collected). When the peer has received all these topology information it needs to be activated. Consequently, the Snapshot Manager will send a request to turn the peer activated which has as an important side-effect to populate the front of its request queue[2] with the in-transit recorded requests. There is no specific per-channel information, as in-transit requests were recorded in the order they were identified, whatever incoming channel they originated from. Hence, the EventCloud is reconstructed, active and ready to receive even additional new requests and serve all of them.

4.2. Characteristics of the supported failure model

4.2.1. Risk of information loss

Let's suppose that the snapshot algorithm instance finishes and suddenly one, some or even all peers in the system crash. From the moment of the crash and the system recovery from the last saved global snapshot, the risk is to miss some of the published events, or some subscriptions. While missing events might be catastrophic if they correspond to critical alerts (e.g. from a nuclear plant), in some other situations such loss may be affordable (e.g. if events are used to signal the arrival of new information to subscribed users of a social network).

Recovering published events is impossible if they were injected in the EventCloud at the following moment: before a crash and no peer was able to serve the corresponding request(s) before it terminated its snapshot taking process. Say differently, there is no effect of the published event on the snapshot constituted by the peer and channel saved states. In practice, it is not realistic to log all events injected into an EventCloud, as it might mean logging the entire actions that happen outside the EventCloud, in the external world possibly at web scale.

Subscriptions might be less numerous than publications. Moreover, when a user submits a subscription, the aim is to be kept informed of matching publications that can happen in the future. A subscription stands as a permanent request until it gets removed

[2]As peer activation is not an atomic process, it could happen that a peer is activated, serves an in-transit request that generates a fresh request that stands in the neighbour queue at the first place, before the logged in-transit ones. It is thus important to force the logged in-transit ones to be put at first, then only activate the peer allowing it to start serving requests: first the logged ones, then the fresh ones. Letting fresh ones bypass recovered in-transit ones would break the FIFO nature of ProActive channels.

by the subscriber. It is thus important to devise a practically feasible[3] solution to prevent any loss of them. The proposition is as follows.

It consists in a subscription-specific automatic retry mechanism, based upon an extension of the existing EventCloud proxies [Roland Stühmer et al. 2011]. In more details, it could leverage the fact that the EventCloud relies upon the assumption that all peers participating in the network run a synchronized clock (this is a reasonable assumption for an EventCloud deployed on an administered infrastructure such as a cluster within a data center). As such, it is possible to tag the last snapshot with the most recent clock value corresponding to the moment all peer states have been fully stored. The proxy is in charge of contacting a peer (known as a tracker in P2P terminology) of the EventCloud, to deliver to it the subscription. Each subscription (and event) entering the EventCloud is gaining the clock value at the moment it enters the EventCloud (because the publish-subscribe matching algorithm requires that information) through the contacted peer. Given that, if an injected subscription gets a clock that is greater than the last checkpoint saved clock value, there is a risk in case of failure, that it gets definitively lost. At the proxy side, knowing the clock value associated to the subscription is easy: the contacted peer sends back an acknowledgment of receipt to the proxy (in the form of either a reply updating the future associated to the initiated method call at proxy side, or as a new request from the peer towards the proxy, as a proxy is also an active object). Obviously if the contacter peer fails before the acknowledgement gets back to the proxy, the proxy could after a timeout automatically retry the delivery, once it detects the EventCloud recovery is over. In all cases, even if not facing a crash of the EventCloud, the proxy could keep a copy of the subscription until being sure there exists a more recent snapshot that involves it. Contacting the snapshot manager allows the proxy to know about the latest saved snapshot clock. Before being sure of that, proxy could regularily reinject the subscription in the EventCloud, not mandatorily contacting the same peer as peer clocks are synchronized. Given each subscription is uniquely identified, any peer receiving it (because it holds a zone that should index events possibly matching the subscription) could thus avoid duplicating it in the corresponding Jena datastore thanks to this unique identifier.

4.2.2. Discussion about partial or whole EventCloud recovery

In search for a better availability of the EventCloud offered as a service to its end-users, one would like to get a fault-tolerant version where only failed peers and not the whole EventCloud is restarted in case of peer failure.

This might be feasible by using a ProActive version implementing the general purpose checkpoint and recovery protocol devised in [Baude et al. 2007]. Indeed, [Baude et al. 2007] proposes a protocol by which a new instance of a snapshot algorithm is triggered periodically. This protocol ensures that no message internally exchanged between active objects is lost even in case of some failure takes place. The main challenge solved by [Baude et al. 2007] is how to correctly recover – only– the failed active objects' state of the system given resulting global checkpoint might indeed correspond to an inconsistent global state. To cope with this challenge and restore the system consistently each

[3]i.e. that avoids systematically logging all information (including publications) that are injected from the external world into the EventCloud

request exchanged in the execution needs to be tagged by the number of corresponding ongoing checkpoint it should belong to. So the need to rely upon a specific "fault-tolerant" version of the ProActive library which today is not usable with multi-active objects. On the contrary, the Chandy-Lamport algorithm adaptation we propose here doesn't assume the reliance upon a modified Active Object programming library: the support of the snapshot messages is simply programmed by extending the by-default request serving policy of active objects. The mentioned request tag is useful in the checkpointing protocol to turn the necessary requests (those identified as violating the global consistency of the global checkpoint) into promised requests: a recovered peer blocks if trying to serve a promised request, until this request gets filled by the data extracted from its replay. These automatic blocking operations and more tricky actions (see details in the paper) eventually ensure the recovered global state of the system becomes globally consistent.

Notice that this fault-tolerant solution suffers from the same limitation pertaining to possible loss of external information sent to the fault-tolerant application (i.e. the EventCloud), as we discussed in 4.1. In fact, if an active object fails before serving a request which could not have the chance to be part of any snapshot, there is no practically feasible solution to recover it even restarting the execution of the application from the very first state, i.e. from the initial state.

Differently from the aforementioned solution, the solution proposed here constructs a succession of global states which are consistent, and from which a direct recovery of the whole system gets feasible.

5. Conclusion and perspectives

In this paper, we have presented an adaptation of Chandy-Lamport Snapshot algorithm that works as a best-effort fault-tolerant solution. To validate the work, the reconstruction of an Event Cloud was made by using the data collected from the snapshot recording and verifying if the reconstructed version had the same features of the initial one (same neighborhood set, Jena data, requests queue, etc).

However some improvements can be done. From a pure performance viewpoint, the snapshot algorithm uses a Java-based library to transfer all the data collected to the safe storage place which could be lowered by using a more native approach.

At this point, our implementation doesn't allow the initiation of multiple instances of snapshot being taken from the same Event Cloud at the same time, that is, to start a new algorithm instance the first one needs to be completely finished. More fundamentally, current implementation assumes that the EventCloud architecture is static that is, peers are not assumed to leave or join the P2P network CAN dynamically. Supporting dynamicity requires not only to modify the implementation, but first to devise a slight variant of the protocol itself. Indeed, a channel in the snapshot taking process is closed once a token from corresponding sender gets served. Allowing the sender to intentionnally leave the network while our distributed algorithm runs may imply no token gets sent from its side. An extension of our proposition could be to enforce a gently leave, i.e. informing the neighbours about the intention to leave before effectively leaving the peer-to-peer network. Symmetrically, allowing a peer to join the CAN network means neighbouring sets of neighbour peers grows while a peer may be already participating in a snapshot taking execution. How to consistently incorporate this new peer state in the global snapshot is

also to be first devised at the protocol level.

Acknowledgements This work received the financial support of the PROGRAMA CIÊNCIA SEM FRONTEIRAS, and the European STREP FP7 project PLAY www.play-project.eu.

References

(2014). Jsch library. `http://www.jcraft.com/jsch/`.

Baude, F., Bongiovanni, F., Pellegrino, L., and Quéma, V. (2011). PLAY Deliverable D2.1 Requirements specification for Event Cloud Component. Technical report, `http://www.play-project.eu/documents/finish/3-deliverables-final/20-play-d2-1-requirements-event-cloud/0`.

Baude, F., Caromel, D., Delbé, C., and Henrio, L. (2007). Promised messages: Recovering from inconsistent global states. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 154–155, New York, NY, USA. ACM.

Caromel, D. (1993). Toward a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

Henrio, L., Huet, F., and István, Z. (2013). Multi-threaded active objects. In Nicola, R. and Julien, C., editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg.

INRIA and UNS (2000). Proactive parallel programming suite. `proactive.inria.fr`.

Jena (2014). Apache jena. `http://jena.apache.org`. Last access made in 1st March 2014.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172.

Roland Stühmer et al. (2011). PLAY Deliverable D1.4 Play Conceptual Architecture. Technical report, `http://play-project.eu/documents/finish/3-deliverables-final/19-play-d1-4-conceptual-architecture/0`.