A Termination Detection Technique Using Gossip in Cloud Computing Environments

Jongbeom Lim, Kwang-Sik Chung, Heon-Chang Yu

A Termination Detection Technique Using Gossip in Cloud Computing Environments

JongBeom Lim[1], Kwang-Sik Chung[2], Heon-Chang Yu[1*]

Department of Computer Science Education, Korea University[1]
Department of Computer Science, Korea National Open University[2]
jblim@korea.ac.kr
kchung0825@knou.ac.kr
yuhc@korea.ac.kr

Abstract. Termination detection is a fundamental problem in distributed systems. In previous research, some structures are used (e.g., spanning tree or computational tree) to detect termination. In this work, we present an unstructured termination detection algorithm, which uses a gossip based algorithm to cope with scalability and fault-tolerance issues. This approach allows the algorithm not to maintain structures during runtime due to node joining and leaving. These dynamic behaviors are prevalent in cloud computing environments and little attention has been paid by existing approaches. To measure the complexity of our proposed algorithm, a new metric, *self-centered message complexity* is used. Our evaluation over scalable settings shows that the unstructured approach can have a significant merit on performance over existing algorithms.

Keywords: Termination Detection, Gossip Algorithm, Cloud Computing

1 Introduction

In the termination detection problem, a set of nodes in the system collectively execute a distributed computation. Determining whether a distributed computation has terminated is a non-trivial task because no node has complete knowledge of the global state, and there is no notion of global time or global memory. Each node only knows its own local state and local time, and communication among nodes can be done only by message passing.

Termination detection problem has been extensively studied for static distributed systems where all of the nodes are stationary in terms of node joining and leaving from the beginning to the end (e.g., [1, 2, 3, 4, 5, 6]). One of systems that termination detection algorithms can be used in is the cloud computing system in which constituent nodes can easily join and leave with dynamic behavior due to loosely-coupled environments. However, although much research for the termination detection problem in recent years mainly focuses on reducing message complexity, little attention has been paid to the aforementioned dynamic behavior. Most of the studies assumed

* Corresponding author.

that the system does not change anymore without considering node failures and joining which are vital aspects in cloud computing environments that should not be dismissed.

Recently, gossip-based algorithms have received much attention due to its inherent scalable and fault-tolerant properties which offer additional benefits in distributed systems [7]. Correctness of a gossip-based protocol is presented in [8, 9]. In gossip-based algorithms, each node maintains some number of neighbors called a partial view. With this partial view, at each cycle (round), every node in the system selects f (fanout) number of nodes at random and then communicates using one of the following ways: 1) Push, 2) Pull, and 3) Push-pull mode. Gossip-based algorithms guarantee message delivery to all nodes with high probability and their variation can be found in [10, 11, 12, 13, 14]. Applications of gossip-based algorithms include message dissemination, failure detection services, data aggregation etc.

In this paper, we present an unstructured termination detection algorithm based on the gossip-based algorithm. The use of the gossip-based algorithm for the termination detection problem is a desired approach to deal with scalability and dynamic behavior in cloud computing systems. Having partial view in the gossip-based algorithm is the essential key to achieve the scalability issue. In other words, each node does not have to maintain all the nodes in the system, but the small number of nodes. Furthermore, in structured termination detection algorithms (using spanning tree or computational tree), reconstruction of the structure of algorithms is needed when node joining and leaving. Otherwise, detecting the termination of a distributed computation is virtually impossible since node connection is broken.

The rest of the paper is organized as follows. We present the system model and formally describe the termination detection problem in Section 2. Section 3 provides our gossip-based termination detection algorithm. Simulation results for the algorithm and their interpretation are given in Section 4; this section also analyzes the message complexity. Finally, Section 5 gives our conclusions.

2 Model and Problem Specifications

2.1 System Model

We assume that the cloud computing infrastructure consists of numerous nodes of resources, and individual nodes process arbitrary programs to achieve a common goal. Because of the absence of shared memory, each process or node should communicate with other nodes only by message passing through a set of channels. In addition, we assume that all channels are reliable and FIFO (first-in, first-out), meaning all messages within a channel are received in the order they are sent to. And the message delay is bounded. There is no global clock. However, it is assumed that each node synchronizes its time by gossiping with other nodes. This approach has been justified by [15]. Furthermore, the communication model is asynchronous. In other words, a sender does not have to wait for acknowledgements of receivers (non-blocking).

2.2 Specifications of the Problem

Termination detection is a fundamental problem in distributed systems; it is not an exception in cloud computing systems. The importance of determining termination derives from the observation that some nodes may execute several sub-problems, and in some cases, there are precedence dependencies among them. Because there is no shared memory, message passing is the only way to deal with the termination detection problem satisfying following properties:

- **Safety**: If the termination detection algorithm announces termination, then the underlying computation has indeed terminated.
- **Liveness**: If termination holds in the underlying computation, then eventually the termination detection algorithm announces termination and henceforth termination is not revoked.
- **Non-Interference**: The termination detection algorithm must not influence the underlying computation.

The definition of termination detection is as follows: Let $P_i(t)$ denote the state (active or passive) of process P_i at time t and $C_{i,j}(t)$ denote the number of messages in transit in the channel at time t from process P_i to process P_j. A distributed computation is said to be terminated at time t if and only if:

$$(\forall i :: P_i(t) = passive) \wedge (\forall i,j :: C_{i,j}(t) = null)$$

2.3 Performance Metrics

Traditionally, the following metric has been used to measure the performance of termination detection algorithms:

- **Message complexity**: The number of messages required to detect the termination.

In addition to the message complexity, we propose a new metric called self-centered message complexity. Self-centered message complexity counts the number of message requited to detect the termination from the requester point of view rather than from the whole nodes in the system. We conjecture that this is more flexible and simpler metric to measure the structured and unstructured termination detection algorithms because some algorithms are not always intuitive and observable at the high-level domain. Self-centered message can be defined as follows:

- **Self-centered message complexity**: The number of messages required to detect the termination from the requester point of view.

3 Termination Detection Technique

In this section, we first review the basic gossip-based protocol based on [16] to describe our gossip-based termination detection algorithm. The termination detection

algorithm proposed in this section can be viewed as an extension of the gossip-based algorithm to support the termination detection functionality.

3.1 Termination Detection Technique Using the Gossip Algorithm

In the gossip-based algorithm, there are two different kinds of threads in each node: active and passive. At each cycle (round), an active thread selects a neighbor at random and sends a message. The active thread then waits for the message from the receiver. Upon receiving the message from the neighbor, the active thread updates local information with the received message and its own information. A passive thread waits for messages sent by active threads and replies to the senders. Afterwards, the passive thread updates its local information with the received message from the sender accordingly.

The function *getNeighbor*() returns a random neighbor identifier from its partial view, not from the entire set of nodes in our algorithm. It is noted that according to the system parameter f (fanout), *getNeighbor*() returns f number of neighbor identifiers. Additionally, before the gossiping is initiated, the partial view of nodes is constructed by the middleware called peer sampling service [16], which returns a uniform random sample from the entire set of nodes in the system.

A simple way to solve the termination detection problem is to use distributed snapshots (e.g., [3]). If a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation. However, the algorithm that uses distributed snapshots broadcasts to all other nodes when a process goes passive; this involves a large number of request messages. Furthermore, detecting whether all the other processes are taken a snapshot is not a trivial job even though all the other processes are passive and taken a snapshot.

Hence, we take the distributed approach with the gossip-based algorithm. To let a process decide whether all of nodes are passive and distributed computation is terminated, we use the piggybacking mechanism by which a node adds additional information of neighbors to the message during gossiping. By using the piggybacking mechanism, any node wishing to detect termination can eventually detect whether distributed computation is terminated or not.

In the previous researches using the distributed approach, however, they assumed that the number of nodes is static. Few studies have focused on the dynamic behavior such as adding and removing nodes while request operations are ongoing, which is that we want to deal with. In the dynamic scenario, it is assumed that each node can learn about newly added and removed nodes by the middleware before each cycle begins.

3.2 Unstructured Termination Detection Algorithm

- *Initial local state for process* P_i
  - ***array of states*** : State$_i$[j] = *passive*, $\forall j \in \{1 \ldots n\}$
- *During gossiping*: Process P_i executes the followings during gossiping with target P_j (where $j \neq i$):
  1. When P_i sends a basic message to P_j:
     (a) $State_i[j].state = State_j[j].state = active$;
     (b) $State_i[j].timestamp = State_j[j].timestamp = LC_{current}$;
  2. When P_j sends a basic message to P_i:
     (a) $State_i[i] = State_j[i] = active$;
     (b) $State_i[i].timestamp = State_j[i].timestamp = LC_{current}$;
  3. Updating states array:
     (a) Update each element of $State_i[k]$ and $State_j[k]$, where $\forall k \in \{1 \ldots n\}$, according to timestamp
- *When local computation is completed*:
  1. Updating local state:
     (a) $State_i[i].state = passive$;
     (b) $State_i[i].timestamp = LC_{current}$;
- *Deciding for termination*:
  1. Checking states array:
     (a) Check if $State_i[k].state == passive$, $\forall k \in \{1 \ldots n\}$
     (b) if (a) is true, then termination is detected.
     (c) if (a) is false, then termination is not detected.

Fig. 1. The proposed unstructured termination detection algorithm

The unstructured termination detection algorithm using a gossip-based approach is summarized in Figure 1. We explain only our extensions to the gossip algorithm. We assume that each process has a unique identifier and can be indexed by from 1 to n, where n is the number of processes (nodes) in the system. Henceforth, the terms a node and a process are used interchangeably.

Each process P_i maintains the following data structures:

- $State_i[1 : n]$: An array of states for P_i. This data structure consists of two elements for each array: state and timestamp. State value can be active or passive and timestamp value is logical time at which state value is updated.

We describe our extensions as follows:

1. If process P_i selects process P_j during gossiping following states are performed:
   (a) When P_i sends a basic message to P_j, State array is updated as follows:

(i) *j*th elements (i.e., state and timestamp) of array of both processes are updated with *active* and $LC_{current}$.

(b) When P_j sends a basic message to P_i, State array is updated as follows:

(i) *i*th elements (i.e., state and timestamp) of array of both processes are updated with *active* and $LC_{current}$.

(c) Each element of *State*[*k*] of both processes, where $\forall k \in \{1 \ldots n\}$, is updated with the one whose timestamp value is larger.

2. When local computation is completed, State values are updated as follows:

(a) $State_i[i].state$ and $State_i[i]$.timestamp are updated with *passive* and $LC_{current}$, respectively.

3. In order to decide whether local computation of whole processes is completed, following states are performed:

(a) State array is checked:

(i) If $State_i[k].state == passive$, where $\forall k \in \{1 \ldots n\}$, then it concludes that termination is detected.

(ii) Otherwise, it concludes that termination is not detected and local computation of some processes is ongoing.

4 Experimental Evaluation

In this section, we present simulation results for the gossip-based termination detection algorithm using the PeerSim simulator [17], which supports extreme scalability and dynamicity of nodes, and is implemented in Java. To compare our algorithms' performance, we implemented the simple broadcast algorithm that is similar to [3], which involve *n*-1 request messages and *n*-1 acknowledge messages. During entire experiments, a fanout parameter *f* is set to 1.

We first evaluated effects of the size of PartialView and the number of nodes. In Figure 2, we can see the results for the requisite number of cycles to detect termination with varying the size of PartialView from 5 to 30 by increments of 5 for 10^4 nodes, and the number of nodes from 10^2 to $10^{4.5}$ with a PartialView size of 20. It is noted that in this experiment, we let each node update its state when a request message is received.

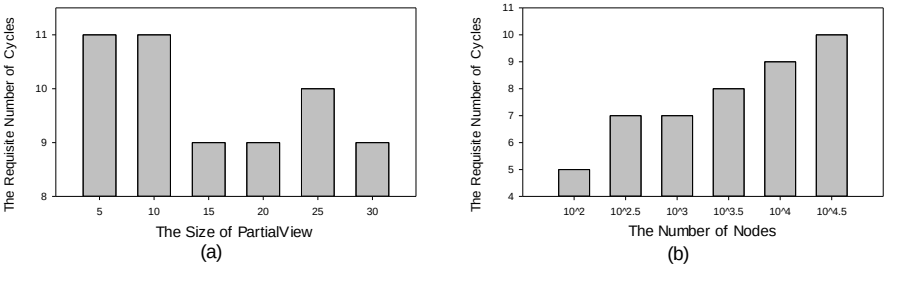

Fig. 2. The requisite numbers of cycles to detect termination. The number of nodes is set to 10^4 in (a). The size of PartialView is set to 20 in (b).

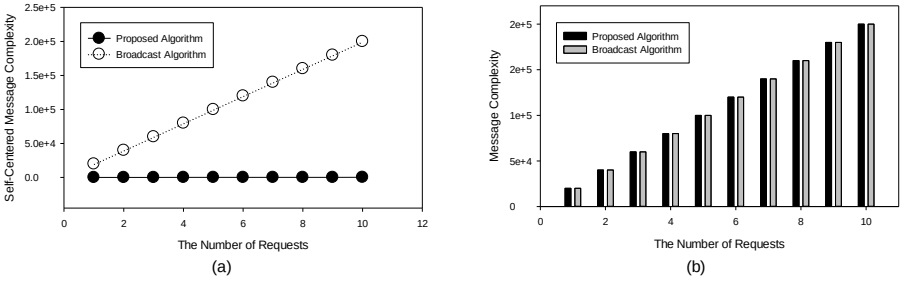

Fig. 3. The comparisons of self-centered message complexity (a) and message complexity (b) between the proposed algorithm and the broadcast algorithm.

We have confirmed that effects of the size of PartialView is insignificant when the size is larger than 15 when the number of nodes is 10^4. Notice that in Figure 2(b), the requisite number of cycles grows linearly as the total number of nodes increases exponentially.

Figure 3 shows the results of message complexity. It is noted that in this experiment, the number of nodes is set to 10^4, and the size of PartialView is set to 20. When we compare self-centered message complexity of the proposed algorithm and the broadcast algorithm, our algorithm performed better than the broadcast algorithm about a 111100% improvement.

During experiments we also have confirmed that if each node changes its state to passive when local computation is finished, rather than when a request message received, the requisite number of cycles to detect termination using our unstructured algorithm is 2. In this regard, when we compare the two algorithms, message complexity is close to each other. In other words, the proposed algorithm generates $2n$ messages, while the broadcast algorithm generates $2(n\text{-}1)$ messages. This signifies that our unstructured termination detection algorithm and the broadcast algorithm have no big difference in message complexity, but in our algorithm, messages are diffused among nodes without a bottleneck.

5 Conclusion

In this work, we have presented the termination detection algorithm using a gossip-based approach to cope with scalability and fault tolerance issues. A cloud environment where the behavior of their constituting nodes is active and dynamic (i.e., joining and leaving at any time) is an example that our algorithm will be applied to. Furthermore, our gossip-based termination detection algorithm could be embedded seamlessly into other existing gossip-based algorithms. In other words, if a gossip-based algorithm is implemented for the failure-detection service, then the termination detection algorithm proposed in our work can be embedded in the existing gossip-based algorithm.

References

1. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters 11, 1-4 (1980)
2. Mattern, F.: Algorithms for distributed termination detection. Distributed Computing 2, 161-175 (1987)
3. Huang, S.-T.: Termination detection by using distributed snapshots. Inf. Process. Lett. 32, 113-120 (1989)
4. Mahapatra, N.R., Dutt, S.: An efficient delay-optimal distributed termination detection algorithm. J. Parallel Distrib. Comput. 67, 1047-1066 (2007)
5. Mittal, N., Venkatesan, S., Peri, S.: A family of optimal termination detection algorithms. Distributed Computing 20, 141-162 (2007)
6. Livesey, M., Morrison, R., Munro, D.: The Doomsday distributed termination detection protocol. Distributed Computing 19, 419-431 (2007)
7. Ganesh, A.J., Kermarrec, A.M., Massoulie, L.: Peer-to-peer membership management for gossip-based protocols. Computers, IEEE Transactions on 52, 139-149 (2003)
8. Allavena, A., Demers, A., Hopcroft, J.E.: Correctness of a gossip based membership protocol. Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pp. 292-301. ACM, Las Vegas, NV, USA (2005)
9. Gurevich, M., Keidar, I.: Correctness of gossip-based membership under message loss. Proceedings of the 28th ACM symposium on Principles of distributed computing, pp. 151-160. ACM, Calgary, AB, Canada (2009)
10. Ganesh, A.J., Kermarrec, A.-M., Massoulié, L.: HiScamp: self-organizing hierarchical membership protocol. Proceedings of the 10th workshop on ACM SIGOPS European workshop, pp. 133-139. ACM, Saint-Emilion, France (2002)
11. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Journal of Network and Systems Management 13, 197-217 (2005)
12. Matos, M., Sousa, A., Pereira, J., Oliveira, R., Deliot, E., Murray, P.: CLON: Overlay Networks and Gossip Protocols for Cloud Environments. Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I, pp. 549-566. Springer-Verlag, Vilamoura, Portugal (2009)
13. Jelasity, M., Montresor, A., Babaoglu, O.: T-Man: Gossip-based fast overlay topology construction. Comput. Netw. 53, 2321-2339 (2009)
14. Lim, J.B., Lee, J.H., Chin, S.H., Yu, H.C.: Group-based gossip multicast protocol for efficient and fault tolerant message dissemination in clouds. Proceedings of the 6th international conference on Advances in grid and pervasive computing, pp. 13-22. Springer-Verlag, Oulu, Finland (2011)
15. Iwanicki, K., Steen, M.v., Voulgaris, S.: Gossip-based clock synchronization for large decentralized systems. Proceedings of the Second IEEE international conference on Self-Managed Networks, Systems, and Services, pp. 28-42. Springer-Verlag, Dublin, Ireland (2006)
16. Jelasity, M., Guerraoui, R., Kermarrec, A.-M., Steen, M.v.: The peer sampling service: experimental evaluation of unstructured gossip-based implementations. Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, pp. 79-98. Springer-Verlag New York, Inc., Toronto, Canada (2004)
17. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on, pp. 99-100. (2009)