2015 IEEE 7th International Conference on Cloud Computing Technology and Science

DCRA: Decentralized Cognitive Resource Allocation model for Game as a Service

Nabil M. Al-Rousan[1], Wei Cai[1], Hong Ji[2], and Victor C.M. Leung[1]

[1]The University of British Columbia, Canada
[2]Beijing University of Posts and Telecommunications
[1]{nabil, weicai, vleung}@ece.ubc.ca
[2]jihong@bupt.edu.cn

***Abstract*— Game as a Service (GaaS) has rapidly emerged to the industry of cloud gaming. The power of GaaS lies on having one source of code with multiple users. Several systems were proposed to model GaaS. However, none has built a scalable and reliable model for such a service. The importance of having such a model lies on having an Internet-scale platform able to provide flexibility of different types of games genre and lower the barrier of end systems (i.e. mobile clients) while taking into consideration the probability of excessive loads and failures. In this paper, we implement a Distributed Cognitive Resource Allocation (DCRA) model to run mobile games on a large-scale distributed system. On the contrary of the existing centralized models, DCRA scales with the increase of mobile clients to handle high concurrent loads of clients' requests while providing a stable level of gaming experience. The results show that DCRA is able to scale well by providing almost fixed throughput and delay while increasing the clients requests load. Also, the system preserve its key features while simulating failures.**

I. Introduction

GaaS has been introduced to mobile game industry to run games on both cloud and clients. Providing the best quality of experience (QoE) to gamers is the top priority. One of the techniques to provide such a service is to implement a cognitive model. A cognitive model redirects game's components to be executed in a cloud and in a client rather than having them all executed in the client or all in the cloud. Cognitive model learns what is the best set of game's components to be executed at the client based on its environment. In this paper, we try to increase the performance of the cognitive model by expanding the client-server model on the cloud side to decentralized system where many nodes on the cloud assist the client to handle the most intense components of the game. Such model should provide no point of failure (fault tolerant). Hence, there will be no bottleneck on one node in the system. Finally, the throughput and round trip time (RTT) would scale better in number of mobile clients and sequentially, the number of off-loaded game's components. This paper is organized as follows: In Section II, we review the related work. The requirements and the design of proposed model are described in Sections III to VI. Evaluation of the proposed mode is discussed in Section VII. We conclude with Section VIII.

II. Related work

GaaS models define a game as a set of inter-connected dependent modules. These modules include *Input*, *Rendering*, and *Game logic* [1]. Different types of GaaS models are defined depending on the allocation of these modules on the client or the cloud sides.

Remote Rendering (RR-GaaS) model is the most popular model where all the modules run on the cloud except the *Input* module. It has been commercially adapted by many companies such as Onlive and Gaikai [2], [3]. The model sends video frames from the cloud to the client through the Internet. Although the client hardware requirements are minimized, network transmission and high cost motivated local rendering on the client side.

To address the problems with RR-GaaS and to benefit from the recent mobile client's hardware advancement, Local Rendering (LR-GaaS) models were developed. The basic idea is to move the *Rendering* module to the client side so the high burden of video frame transmission is eliminated. However, finding an instruction set to transform all the games visual frames through the Internet to the mobile client is an unresolved research problem [1].

To overcome the problems in both RR-GaaS and LR-GaaS, Cognitive Resource Allocation (CRA-GaaS) was introduced [4]. The model keeps the *Input* and *Rendering* modules on the client side but divides the *Game logic* between the cloud and the client. CRA breaks a game into inter-connected dependent components. The cognitive ability allows the CRA model to optimally and dynamically finds the best selection of components to run on the cloud to increase the Quality of Experience (QoE) for the end users.

From software engineering perspective, CRA model is an API designed for game developers to develop games in such a way that the game breaks in execution to several components run both on the mobile client and on the cloud. The current CRA design allows the game's developer to decompose the game code into components by marking peaces of the code into different component IDs. Decomposition problem is an open research area and dynamic decomposition is the ideal solution where components migration to the cloud depends on the client and server state.

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.63

All the previous models share the features of GaaS models which include: click-and-play, anti-piracy, development cost reduction, and cross-platform gaming experience. Game Genre plays an important role in the selection of the GaaS model. For example, 1st-person shooting games where the scene images changes on a high rate work best with RR-GaaS since the scene variety and motion frequency are very high. On the contrary, 3rd-person management games where the scene variety and motion frequency are low, fit CRA-GaaS since the game logic is the most intense module such games. Current browser games fall into LR-GaaS where all the rendering occurs on the client side.

Although CRA seems to be the next GaaS generation, the current design is a client-server design. As a result, it lacks the features of a real distribution system such as scalability and reliability. In a client-server model, excessive load on the server can slow down or break the system. In such a system, the server acts as a bottleneck on the system.

From the software engineering perspective, centralized CRA lacks dynamic component execution where the components are argument-ed and do return a value. As shown in Fig. 1a, the current CRA model takes user input only once before executing all the three components. However, in (Fig. 1b), the game is able to receive user input while executing components 3, 5, and 4 on the cloud. Also, component 5 depends on the output of game 3 and the user input. Hence, we expanded the cloud side implementation of CRA to replace the client-server implementation to decentralized one and therefore, there is no point of failure. DCRA overcomes the problems of excessive load in CRA by building an overlay network in the application layer to distribute the tasks of CRA. DCRA works under the assumption that there is a stable connectivity between the mobile device and the cloud with no interruptions. Dealing with temporary disconnections are left for future work.

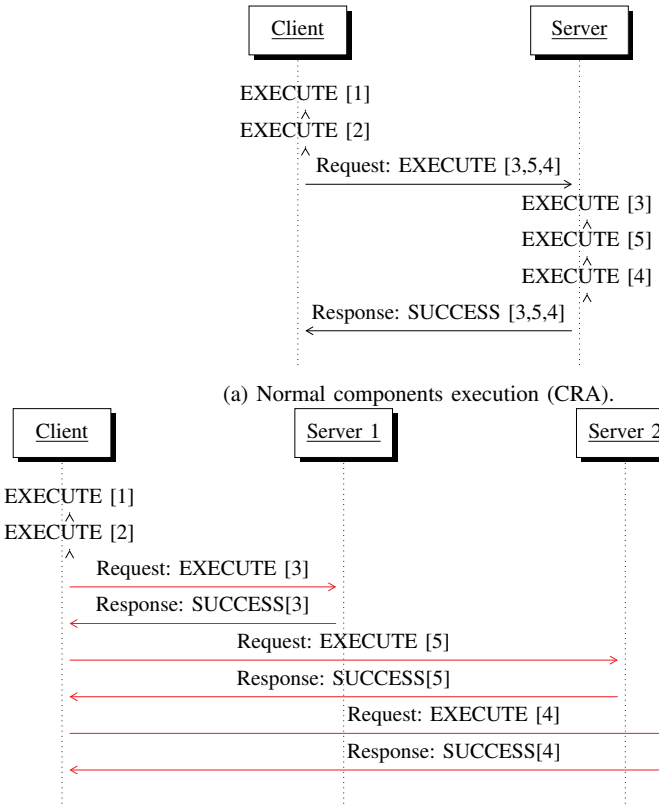
III. System Objectives

DCRA is built over an P2P overlay on the cloud. several problems arise with such design choice. The biggest concern is RTT delay. Games are very sensitive to RTT as it the most important factor in the QoE requirements. However, we have implemented various techniques to reduce the delay as much as possible. These include membership protocol and caching. Also, The typical life-time of any server in the cloud is unpredictable and usually uncontrolled by the system designer. Hence, reliability is a concern in such an environment. Lost or delayed requests result an immediate game pause or even a game crash. Also, uneven distribution of work load by some popular component might slow down the system. To address these problems, we have designed DCRA to achieve the following objectives:

- Scalability: The system should scale out with the increase of the number of mobile clients in terms of throughput and RTT. We have applied various techniques to achieve scalability. First, distribution of single computation task and spreading it across the node servers in the cloud. Hence, avoiding single server to deal with all clients requests. Second, by using asynchronous communication which hide the communication latency. This allows the system to continue serve tasks while waiting for a server's reply for some other requests. Third, Caching. Servers cache the replies for all the processed requests for certain period of time. In case a request was lost, a server, A, will resend the request again to server B. Since B has the reply cached for A's request, it will not deliver the request to the upper layers. Instead, it will serve it from the cache. More details in V.
- Availability (Reliability): The system should be always available. In other words, every request of any mobile client should receive a reply from a server. Achieving availability is very hard in distributed system in the face of failures and network delays. We overcome these failures by replicating the execution of mobile client over a set of server nodes. This technique is known as *Process Resilience*. The general idea is to mask process failure by replicating the execution of a single process over a group of servers (resilience group). Another advantage of *Process Resilience* is to obtain better performance in terms of latency. A mobile client access time will be reduced since it receives the the first reply from any member of the resilience group.
- Fault Tolerance: The system should be tolerant to failures. Many types of failures can affect the availability of the system. The most common is the Fail-stop failure where a server crashes and the other servers detect its failure. To overcome this type of failures, we need to detect the failure first. We achieve this by implementing a membership protocol so each server have a global view of all the other servers state in the system. Second, availability should be preserved regardless if a set of servers are up or down. Again, we solve this issue by *Process Resilience* by making sure that there is at least one server node to respond to any client request. Other types of failures are Byzantine failures which are arbitrary failures where servers or mobile clients might produce malformed requests or replies. In gaming context, cheating requests might be sent from clients to servers. We do not cover such type of failures in this scope and we leave it as a future work.
- Heterogeneity: The system should detect heterogeneity in hardware specs of different mobile clients and network bandwidth for the infrastructure that they are running on. We plan to achieve this objective by running reinforcement learning as partitioning algorithm to specify how much computation should be offloaded to the cloud. Client side API is left as future work.

IV. DCRA: Network Design

To achieve the previous objectives, a novel decentralized distributed system is designed and implemented. The system is a collection of server nodes on the cloud that appear as a single coherent system to the mobile client. The server nodes group together to distribute execution of one task from



(a) Normal components execution (CRA).

(b) Dynamic components execution (DCRA).

Fig. 1: Execution of Game Components.

the mobile client perspective. Hence, Coordination between nodes is needed to achieve correctness of the proposed system.

A. Game decomposition

Every game is decomposed to set of dependent components. In a game context, a component is a building block that differs from other components in functionality [7]. Fig 2 illustrates the partitioning of a game components between the mobile client and cloud. For example, component 3 execution depends on component 2 execution. Each component is stateless where no global variables are shared. For the previous example, component 3 takes component 2's output as an input and execute the code of its own before sending its output as an input to component 5. By supporting dynamic component execution, it makes no difference for the client if components 3, 4, and 5 where executed on the same server node or in different ones. In both ways, it is one hop distance to the client so there is no extra distance for the client request to forward.

B. Overlay Network

From networking perspective, DCRA is an overlay network over set of physical nodes on the cloud. A virtual ring topology constructed to assign an incremental ID for each server. The ID range is from 0 to $N-1$ where N is the

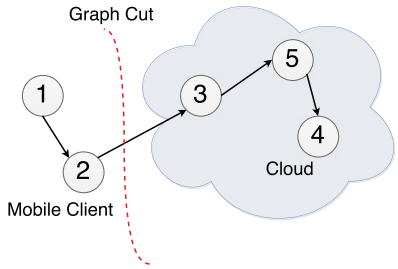


Fig. 2: partitioning of game component between the mobile clients and the cloud.

total number of server nodes. We have tested our system on a 95 nodes on PlanetLab [10]. PlanetLab is an open global research network that is consisting of 1337 nodes around the world. Although, we could have used more nodes to test DCRA, a quick sanity test is performed on the 1337 and showed that 95 are functioning correctly.

Fig. 3 illustrates the naming of the server nodes in the overlay. For the game shown in Fig. 2, component 1 and 2 are executed on the client side. The rest are executed in the

cloud by sending EXECUTE commands from the client to a random server node.

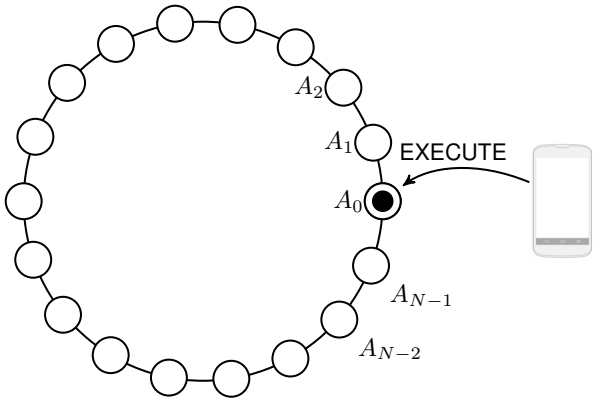


Fig. 3: DCRA as an overlay network.

C. Membership protocol

A Gossip-based membership protocol is implemented to obtain a global view of the state of each node in the system [11]. We could have replaced the whole membership protocol by a simple probe PING message before any one-to-one communication but that would add an extra overhead and delay to the system. Taking into consideration the requirements of GaaS and the objectives of the system, a membership protocol had to be implemented to have the aliveness state of any server node in the system at any time instantly.

To implement a membership protocol, each node holds an $aliveness$ table with $node_id$, $t\_last\_updated$, and a $heart\_beat$ counter. Regularly, each node sends HEARTBEAT messages to $log(N)$ set of random nodes by a gossip algorithm (Algorithm 1). Upon receiving a HEARBEAT message, a node updates the local $aliveness$ table by increasing the $heart\_beat$ counter by one and setting the $t\_last\_updated$ to the current local time.

```
repeat
    count = 0;
    repeat
        count ++;
        send message to random node;
    until count == log(N) nodes;
until random number <= 1/k;
```

Algorithm 1: Gossip Algorithm.

where k is a constant chosen arbitrary (4).

For node Y, to check if node X is alive, the current condition must hold true:

$$time.time() - t\_last\_updated < T\_Fail \tag{1}$$

where $time.time()$ is the current local time and T_Fail is a constant set arbitrary to 3 seconds. The condition ensures that if a node X is alive, there should be an update for X within the last 3 seconds. Node X can be updated by HEARTBEAT messages from itself or by a DISTRIBUTE message from another node.

The gossip algorithm sends DISTRIBUTE messages periodically (set to 3 seconds arbitrary) to $log(N)$ nodes. The DISTRIBUTE message contains the local $heart\_beat$ counters for each node in the $aliveness$ table. Upon receiving a DISTRIBUTE message, the local $heart\_beat$ counter for a node is updated (by increasing the $heart\_beat$ counter by one and setting the $t\_last\_updated$ to the current local time) only if Eq. 2 holds true. The upper part of the condition guarantees that the remote update is more recent than the local one. The lower part prevents the oscillation in the state of a node when it dies.

$$\begin{gathered}[remote\_HEARTBEAT > local\_HEARTBEAT] \\ AND \\ [(time.time() - t\_last\_updated < T\_Fail \\ OR \\ T\_Clean < time.time() - t\_last\_updated)],\end{gathered} \tag{2}$$

where T_Clean is defined as:

$$T\_Clean = 2 \times T\_Fail. \tag{3}$$

The oscillation might occur when a node W dies but two other nodes keep incrementing the $heart\_beat$ counter for W node continuously upon receiving DISTRIBUTE messages from each other containing a $heart\_beat$ higher than the local one. To prevent such a scenario, the lower part of the condition guarantees that for a period of time more than T_Fail and less than T_Clean, any of the two nodes will not accept an update for the W node. This also means that after T_Clean there will be no updates for dead node unless it comes alive again since the only way to update its status is to have a new HEATBEAT messages sent by W itself.

D. Routing/Naming

Routing of a game's component is achieved by consistent hashing [12]. A key is defined as the hash of the concatenation of $GameID$ and $ComponentID$. Each key is assigned a server node by

$$hash(key)\%N \tag{4}$$

where N is the total number of nodes. This will result a uniform distribution from 0 to $N-1$. Consistent hashing ensures load balancing. Hence, If there is K keys distributed across the system, consistent hashing ensures that every node is assigned K/N keys. Although all keys are evenly distributed across the system, some popular games might translate into highly executed keys. We leave this problem as future work, however, it can be seen from the Eq. 4 that by adjusting the keys concatenation into an independent game string value, a more efficient load balancing can be achieved.

Each node is assigned an ID between 0 and $N-1$. Naming is necessary so each node can identify other nodes. This allow mapping between the node ID and the node

address (IP and port number). General naming solutions can prevent systems from scaling, but using consistent hashing, the complexity of looking up a node is $O(1)$. Traditional techniques are client-server (Napster), broadcasting (Gnutella), and DHT (BitTorrent) [13]. The client-server and broadcasting techniques proved not to scale. DHT look up needs $O(logN)$ operations to find a node which is still worst than $O(1)$. The only drawback for consistent hashing as method for Naming is the space complexity for routing state stored in each node. The space complexities are $O(N)$ and $O(logN)$ for consistent hashing and DHT, respectively. However, for a system with hundreds of nodes, the space complexity difference is negligible.

E. Process Resilience

Process Resilience for functions is what replication for data. It aims to achieve availability and fault tolerance. To protect against process (nodes) failures, we organize several identical processes to run in parallel in a resilience group with a resilience factor (R_f) of 3 (two resilient processes plus the original process). The resilience group is dynamic and determined with the help of the membership protocol. It is set by finding the next two alive nodes on the counter-clock wise direction of the ring. The purpose of this operation is to abstract the execution for the client. The client does not know how many server are executing its function or which server is replying back. Fig. 4 illustrates the execution of one game component where a node sends 3 EXECUTE requests to 3 alive nodes and wait for N_E replies. N_E is constant set by default to 1 but it can be adjusted by the game developer ($1 <= N_E <= R_f$) to control the Quorum size. If it is set to three, Node A_0 has to wait for all the three replies from nodes 5, 7, and 8. Hence, better consistency but less availability.

Back to the game shown in Fig. 2, components 3, 5, and 4 are distributed based on the consistent hashing to three server nodes. Every server replicates the received EXECUTE to R_f other nodes by sending HINTED_EXECUTE. Next, it waits for N_E instead of R_f replies to minimize the latency. We have also tried to send the replies directly to the mobile client by also sending the client address along with the EXECUTE and HINTED_EXECUTE messages to cut the latency even more. However, due to UDP's socket security, the routers in-between drop any UDP reply with a different receiver (now sender) address.

V. DCRA: Software Design

DCRA is implemented fully in Python. To aid the design, development, and troubleshooting, the implementation is divided to service layers where each layer add specific functionalities to the system:

1) UDP layer:
   Basic UDP sender and receiver are implemented in this layer. UDP is chosen over TCP to avoid the TCP connection setup/termination delay. UDP unreliability is masked by using a Request-Reply layer and by using *Process Resilience*.

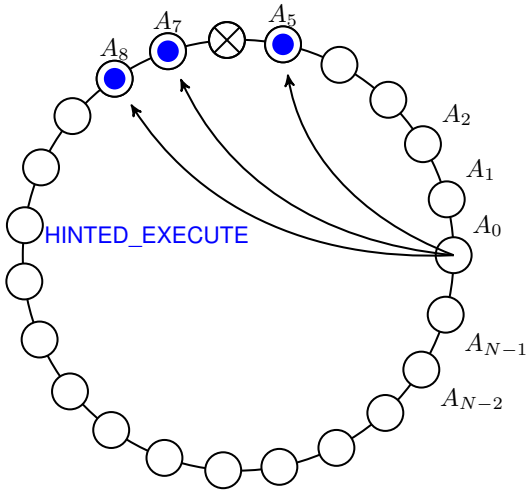


Fig. 4: Process Resilience: Node A_0 sends R_f EXECUTE requests.

2) Request-Reply layer:
   This layer adds some reliability to UDP by implementing request-reply protocol over UDP. Three retries after the first request will be sent to the receiver before considering it down. This would ensure some level of reliability over UDP. A Client will add a unique header in front of each request message to identify the request's reply from the server. The identifier is generated using local time, IP, and port combination so it will be always unique. Consequently, the server will use the same unique header in its reply so each reply is paired to a request.
   A cache is implemented on the Server side of the protocol to hold the replies for 5 seconds in case of a duplicate message was received. A duplicate message can be detected by the unique identifier field by searching the cache. The client timeout can be set also by the application developers.
3) Wire layer:
   This layer is implemented on top of the Request-Reply layer to add the syntax for application-level commands. Some of the used Commands
   - HEARTBEAT: used by the membership protocol to send alive messages to the destinations
   - DISTRIBUTE: used by the membership protocol to share the local $aliveness$ table
   - EXECUTE: Used by a client to execute a function on a server node.
   - HINTED_EXECUTE: Used by a server node to hint off the execution to a proper node using consistent hashing
4) Main layer:
   This layer use the Wire layer's commands to build the logic of the system. Each server nodes basically runs an infinite loop waiting to receive a a request which it will offload to a separate queue upon receiving to be

processed. Hence, no requests will be dropped. To ensure the asynchronous communication, R_f queues are used as buffers for the process resilience operations. At the same time, R_f threads always check for any tasks pushed in the queues. If any found, the threads pop the tasks and serve them until there is no more tasks to be done. Hence, The main loop is always available to receive a new requests which increases the availability of the system.

VI. Fault Tolerance

Although hardware failures probability is too low, it is still a major factor in distributed system design in a large-scale systems [14]. DCRA handles Fail-stop failures only. Fail-stop failures are the most common type of failures in which a server node stops to respond. Fail-stop failure can be detected by time-out event while waiting for a request's reply. To make the system tolerant to such failures, it has to detect and mask the failure. DCRA detects the failure by using the membership protocol. The system is $K + 1$ tolerant meaning that it can tolerate K failures for $N = K + 1$ servers. Hence, one server node is sufficient to run the system. Masking failures mechanisms depend where the failure may happen:

1) Server node failure:
   At-least-once semantic: The system guarantees it will carry out an operation at least once which is maintained using *Process Resilience*. (*exactly-once* and *At-most-once* semantics are the other design variants). From the nodes in the ring perspective, the failure can occur:
   - After HINTED_EXECUTE command was sent: The node will receive the reply from other member nodes in the resilience group.
   - Before HINTED_EXECUTE command was sent: The request will be sent to the successor of the failed node using the membership protocol.

   From the clients perspective: The request will timeout and a new random server node will be selected.
2) Client node failure: The server is executing and holding system resource (Orphan computation). After rebooting, client kills all of its processes (KILL command). The Cache on the server side will help to reduce the duplication of processing client requests

As for masking message loss:

1) Request message is lost: Client's request will time out and retry request will be sent with the same unique ID of the original request. Even if the three retries failed, the EXECUTE command is always idempotent which means that it is repeatable without any harm done if it happened to be carried out before.
2) Response message is lost:
   - If some replies are lost: Do nothing, other server nodes replies will be received from the members of the resilience group.
   - If all replies are lost: Retries with unique request IDs (idempotent EXECUTE).

VII. Evaluation

Various tests were performed to verify the ability of the system so scale while preserving the requirements of GaaS. The tests simulate running games by sending EXECUTE commands to the system and observe the statistics of round trip time and throughput. For simulating purposes, each game's component is simulated by a loop of 1,000,000 iterations.

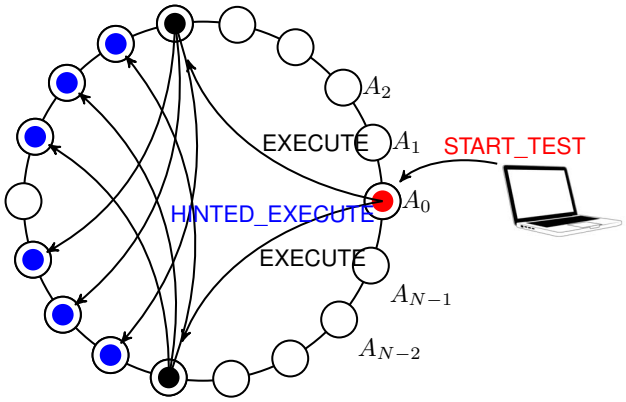
A. Performance tests

This objective of the test is to measure performance on large scale deployment. The test reports the response time and throughput of the system with low, medium, and high load. In this test, an observer node sends START_TEST request to multiple nodes on the system to start a test with specific parameters. The parameters include: number of EXEUCTE requests need to be send, retrial times for each EXECUTE request, pausing time between any two EXECUTE requests, timeout for each EXECUTE request, and the number of nodes to send the EXECUTE requests to. The test has two variants:

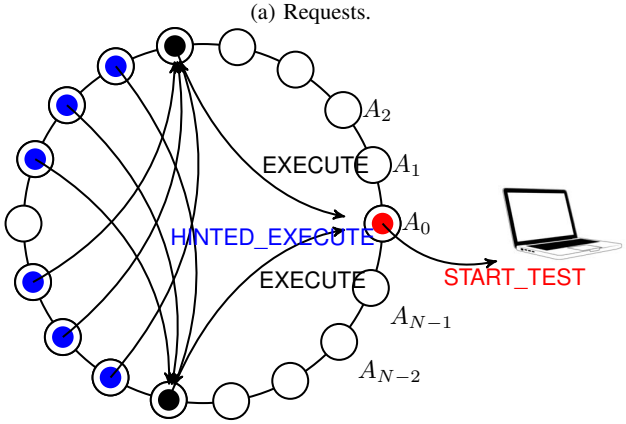
1) Multiple nodes test: In this variant, the EXECUTE commands are sent to other random nodes in the system as shown in Fig. 5. Fig. 6 illustrates the average RTT for various tests. It is shown that the RTT scales with the increase of the number of different nodes. However, it decreases as the number of nodes increases. This shows that the cumulative RTT is saturating at a fixed value. Cumulative RTT is the multiplication of the number of messages with average RTT. To understand scalability in Fig. 6, the cumulative RTT would give almost a linear function in which the x axes is the increase in the number of messages and y axis is the commutative RTT.

To have a better understanding of the RTT delay, it takes around 115ms on average to execute the EXECUTE command locally on any server node in the system. Adding 100-250ms for ICMP typical ping delay [15]. The 440ms sounds an acceptable delay from the system perspective. As for the user experience, the 440ms falls into the QoE requirements for DCRA's games genre.

2) Single node test: The previous test simulates DCRA in normal operation. DCRA acts as CRA in the worst case in which there is only one server node in the system. To simulate CRA, we oberved the performance while applying high load on a single random node. As shown in Fig. 7, Multiple server nodes simulates the client nodes and send EXECUTE requests to the single node. It is shown in Fig. 8 that the performance is worse than the multiple nodes test (Fig. 6) as the number of nodes increase. This is expected since all the EXECUTE commands are sent to one node in the ring. Hence, filling the threads queues with a high number of tasks. This test also aims to verify the correctness of DCRA in the worst case.

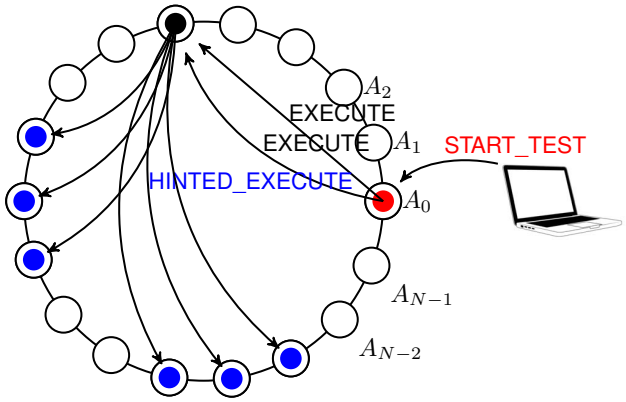


(a) Requests.

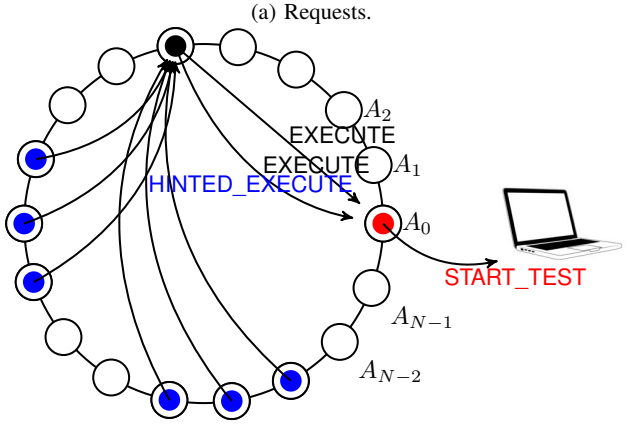


(b) Replies.

Fig. 5: Multiple nodes test.



(a) Requests.



(b) Replies.

Fig. 7: Single node test.

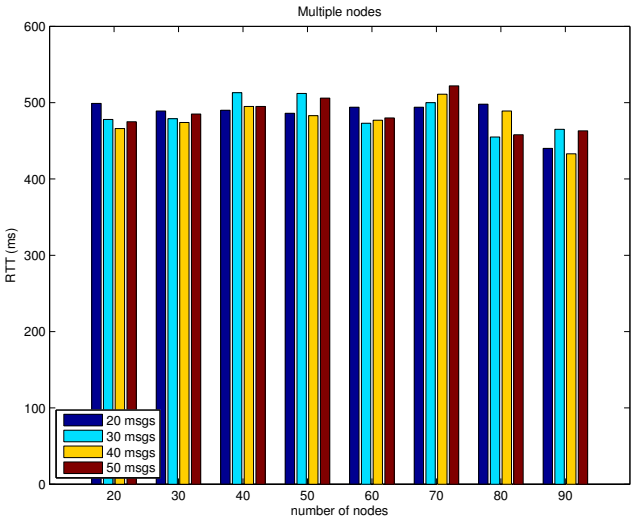


Fig. 6: RTT for multiple nodes test.

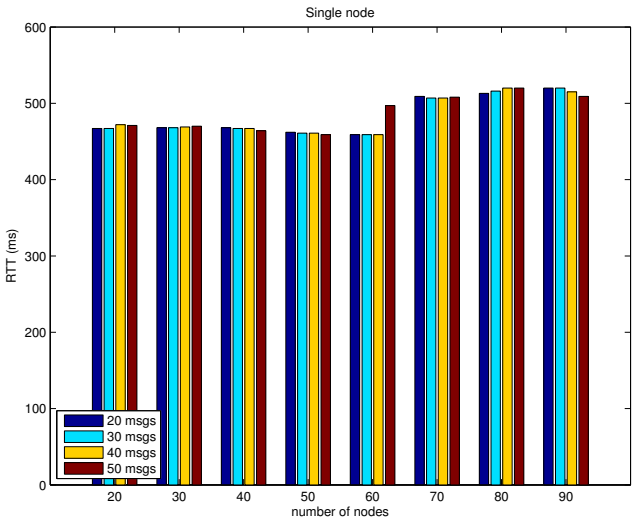
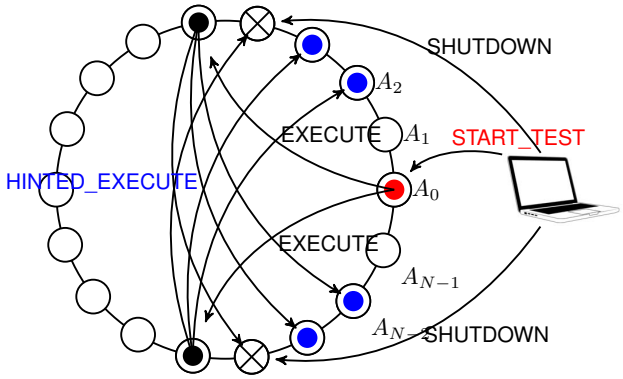


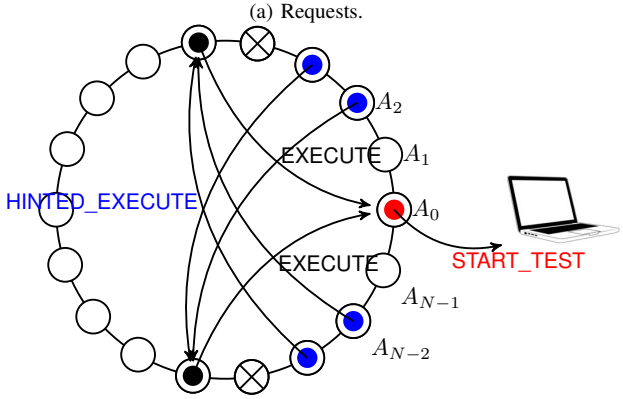
Fig. 8: RTT for single node test.

B. Catastrophic failure test

To verify the availability and the reliability of the system, A SHUTDOWN request is sent to 20% of the server nodes to simulate a catastrophic failure while sending EXECUTE requests to multiple nodes. The test should result no significant drop in performance in terms of response time and throughput. Fig. 9 illustrates the scheme of the test. As shown in Fig. 10, the RTT decreased after the failure, however, DCRA still function correctly. Hence, proofing the high convergence of its membership protocol. The system cover



(a) Requests.



(b) Replies.

Fig. 9: Catastrophic failure test.

fail-stop failures where nodes stops responding expectingly. Other failures like Byzantine failures are not masked by the system. They are left for future work.

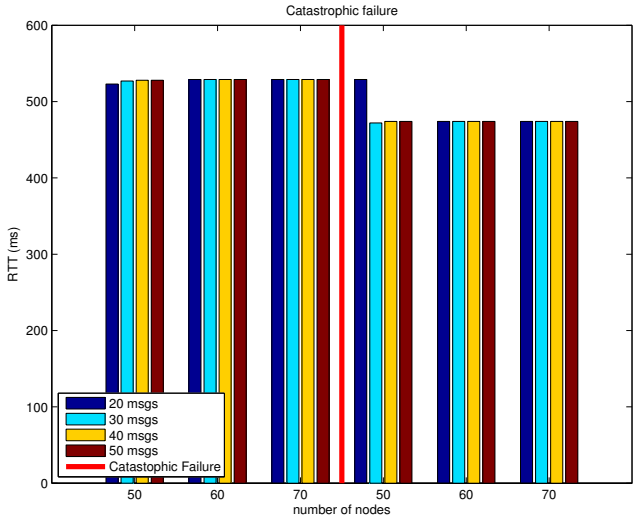


Fig. 10: RTT for catastrophic failure test.

VIII. CONCLUSION

We have implemented a decentralized cognitive resource allocation system to host a platform to run games over the cloud. The system matches the required objectives: availability and fault tolerance while maintaining scalability by applying various state of the art techniques. The system shows stable performance on high load with an average RTT around 400ms (2.5 requests per second) for a single mobile client. The systems performance evaluation shows that DCRA succeeds CRA in terms of throughput and RTT. Finally, the system shows that with proper design choices, mobile gaming can benefit from the distribution of system services and use DCRA in real world mobile gaming on a large scale deployment.

ACKNOWLEDGEMENT

This work was supported by a University of British Columbia Four Year Doctoral Fellowship, the Canadian Natural Sciences and Engineering Research Council under grant STPGP 447524-13, and the National Natural Science Foundation of China through Project 61271182.

REFERENCES

[1] W. Cai, M. Chen, and V. C. M. Leung, “Towards Gaming as a Service,” *IEEE Internet Computing*, vol. 18, no. 3, May/June 2014, pp. 12–18.
[2] OnLive [Online]. Available: http://onlive.com/
[3] Gaikai [Online]. Available: https://www.gaikai.com/
[4] W. Cai, C. Zhou, V. Leung, and M. Chen, “A Cognitive Platform for Mobile Cloud Gaming,” presented at IEEE International Conference on Cloud Computing Technology and Science, Bristol, UK, December 2013.
[5] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, "Clonecloud: elastic execution between mobile device and cloud", in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys’11, ACM, New York, NY, USA, 2011, pp. 301–314.
[6] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In Middleware, 2009.
[7] W. Cai and V. Leung, ”Decomposed Cloud Games: Design Principles and Challenges”, in *IEEE International Conference on Multimedia and Expo (ICME2014)*, Chengdu, China, July 2014
[8] C. Xu, J. Rao, and X. Bu., "A unified reinforcement learning approach for autonomic cloud management", *Journal of Parallel and Distributed Computing*, vol. 72.2, pp. 95-105, 2012.
[9] S. Singh, P. Norvig, D. Cohn, and H. Inc, *"How to Make Software Agents Do the Right Thing: An Introduction to Reinforcement Learning"*. Adaptive Systems Group, Harlequin Inc, 1996.
[10] PlanetLab | An open platform for developing, deploying, and accessing planetary-scale services [Online]. Available: https://www.planet-lab.org
[11] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Service,T Proc. Conf. Middleware*, pages 55–70, 1998.
[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web", in *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC ’97. ACM Press, New York, NY, 654-663.
[13] A. Tanenbaum, M. V. Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007
[14] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote from LADIS, 2009.
[15] IP Latency Statistics [Online]. Available: http://www.verizonenterprise.com/about/network/latency/