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

by

Nabil M. Al-Rousan

B.Sc., Jordan University of Science and Technology, 2009
M.Sc., Simon Fraser University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

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

submitted by **Nabil M. Al-Rousan** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Sathish Gopalakrishnan, Electrical and Computer Engineering
Supervisor

Karthik Pattabiraman, Electrical and Computer Engineering
Supervisory Committee Member

Purang Abolmaesumi, Electrical and Computer Engineering
Chair

Abstract

Gaming-as-a-Service (GaaS) has rapidly emerged to the industry of cloud gaming. The power of GaaS lies on having one source code base with multiple users. Several systems were proposed to model GaaS. However, there are no scalable and reliable models 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. We present a Distributed Cognitive Resource Allocation (DCRA) model to run mobile games on a large-scale distributed system in which we have improvised a unique distributed hash table (DHT)-based routing to expedite the messaging among servers and to minimize the round trip delay to acceptable levels for the targeted mobile games genre.

In contrast to 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.

Lay Summary

Playing console and PC games on mobile phones has been an industry demand in the last few years. However, mobile devices' hardware and power consumption limitations have made the process of migrating desktop games to mobile devices difficult. We have designed a platform to mediate the challenges of handling the complexity of migrating a resource intensive games design to devices with limited processing capabilities by using the aid of remote servers (in the cloud).

Preface

This dissertation is original and independent work by the author, Nabil M. Al-Rousan. A version of this material has been published as N.Al-Rousan, W. Cai, H. Ji and V. Leung, "DCRA: Decentralized Cognitive Resource Allocation Model for Game as a Service", *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

Game decomposition algorithm in Section 1 from Chapter 2 was done primarily by W. Cai. The construction of DCRA's network and software models in Chapters 2-4 was done by my me. Dr. Sathish was the supervisory author on this project and was involved throughout the project in concept formation and manuscript edits.

Table of Contents

List of Tables

List of Figures

Acknowledgements

This work would not have been possible without the support of Dr. Sathish Gopalakrishnan. I am especially indebted to Dr. Sathish who have been supportive of who worked actively to provide me with the protected academic time to pursue my studies.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my mother, whose love and guidance are with me in whatever I pursue. Most importantly, I wish to thank my loving and supportive wife, Amani, and my three wonderful daughters, Maryam, Asmaa and Ayah, who provide unending inspiration.

Chapter 1

Introduction and Review

1.1 Motivations

Mobile games have recently shared a broad market base due to its competitive edge advantages such as context awareness, ubiquity, and personalisation [1]. Mobile gaming changed the gaming experience from offline single player gaming to online large scale multiplier gaming. However, due to mobile clients limited hardware specs compared to gaming consoles, mobile gaming suffers from running rich resource intensive games that needs huge memory and graphic processing.

To overcome this limitation, Gaming-as-a-Service (GaaS) has rapidly emerged to the industry of mobile gaming by using the Internet cloud. This approach eliminates the need of the mobile client to execute intensive game components and extend the traditional mobile gaming model with additional components on the cloud. Several systems have been proposed to model GaaS. However, there is no scalable and reliable model for such a service. The importance of having such a model lies on using 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. We propose to design, develop, and implement a Distributed Cognitive Resource Allocation (DCRA) model to run mobile games on a large-scale distributed system. In contrast to 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 early 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.

We also provide an abstraction to simplify building mobile games in a decentralized edge cloud setting. We present a language support that allows developers to recast a mobile game design to its core functionality without the need to build a full stack of distributed services from the ground up.

1.2 Research Objectives

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 thesis, 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.

DCRA is built over an P2P overlay on the cloud [2]. 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. Hence, we have implemented various techniques to reduce the delay as much as possible. These techniques include membership and caching protocols. 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:

- Quality of Experience: The system should provide the user with QoE properties: Responsiveness, fairness, and precision [3]. All these properties determine game playability and hence a certain QoE level. Authors in [4] have defined these properties as the following. Responsiveness reflects the time the system need to act upon a user interaction. It give the user sense of perception while playing the game. Fairness is a measure of consistency among the various players in game's state for the same game session. Finally, Precision is a measure of the state difference value between a client and server game states. All of three measures should be maximized to obtain a better QoE level.

- Scalability: GaaS models vary in their architecture as distributed sys-

tems. To our knowledge, no GaaS model has solved the problem of scalability. Also, the fact that users and resources are far apart, requires the system to geographically scale. The system should scale out gracefully with the increase of the number of mobile clients in terms of throughput and RTT. Finally, the model should administratively scale while spanning multiple independent organizations. If scalability is not featured for any distributed system, there will be often a loss in performance while the system scale up [8]. 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 serving 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 Chapter 3.

- Availability (Reliability): In GaaS models, resources are shared in the cloud. With resource sharing, a client will request to access these resources at the same time, The system resources 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 [9]. We overcome these failures by replicating the execution of a 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 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. Secondly, 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.

- Programmability: GaaS involves thousands of entities whose location and behavior might vary thought out the lifetime of the system. Given the changing conditions of the edge resources, such service demand good interactive response and consistent data sharing. To meet the challenges of decentralized cloud nature, we provide a middleware that mainly support*multi-consistency decoupled-coordination* communication. We claim that it is the right abstraction for edge interactive applications in a decentralized edge cloud setting. We also present language support that allows developers to recast an edge application design to its core functionality focusing mainly on the game logic.

1.3 Background and Literature Review

GaaS models define a game as a set of inter-connected dependent modules. These modules include *Input*, *Rendering*, and *Game logic* [10]. 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 [11, 12]. 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. Authors in [13] study the effect of reducing the transmitted images bitrate

over the QoE of the GaaS system. Although is it shown that the layered coding performs better than direct coding in terms of QoE, however, the cost of such change will affect the quality of the transmitted frames significantly. RR-GaaS models suffer from finding a green energy solution for the client complex computations without compromising the QoE of the user.

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 [10].

To overcome the problems in both RR-GaaS and LR-GaaS, Cognitive Resource Allocation (CRA-GaaS) was introduced [14]. 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 for the end users.

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 [15]. 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 Figure 1.1a, the current CRA model takes user input only once before executing all the three components. However, in (Figure 1.1b), 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 implementa-

tion 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.

Different studies have defined and evaluated gaming QoE for the current GaaS models [16–19]. In [16], the authors designed and tested an Ad-Hoc Cloudlet GaaS system and tried to enhance the user QoE by optimizing game video frames. Although, the trace driven simulation shows better QoE, the paper did not solve the general RR-GaaS problems. In [19], the authors propose "Cloud Fog" distributed system in which RR-GaaS systems may offload high intensive computations to nearby nodes on the cloud. Hence, enhancing QoE. The results were promising, however, authors showed with preliminary results that optimization of geographic selection of servers is still an open problem. In [17], authors tried to boost QoE using Lagrangian Relaxation (LR) time-efficient heuristic algorithm. The algorithm outperform the existed server-centric and network-centric in which the selection of server is decided by processing power delay and network delay, respectively. The authors, however, did not investigate large scale deployment of there model. In [18], the authors of [19] continue their work to enhance QoE for Massively Multiplayer Online Games (MMOG). The authors applied the concepts of caching and super nodes on the cloud to reduce the downstream delay time as much as possible. The fixed requirements of such system requires stable super nodes and does not guarantee fault tolerance against communication failures.

To our knowledge, all the previous works lack the seamless abstraction between the GaaS model and the developer code. The idea of designing abstractions to make the process of implementing distributed systems is not new. However, to the best of our knowledge, the research presented here is the first to propose a general middleware for *multi-consistency and decoupled coordination* communication for mobile gaming applications. We believe that operation-based consistency along with runtime support is necessary for such an abstraction. Of course, we derived inspiration and borrowed many ideas from other researches. We hope that this work will influence future research into designing better features for edge middlewares.

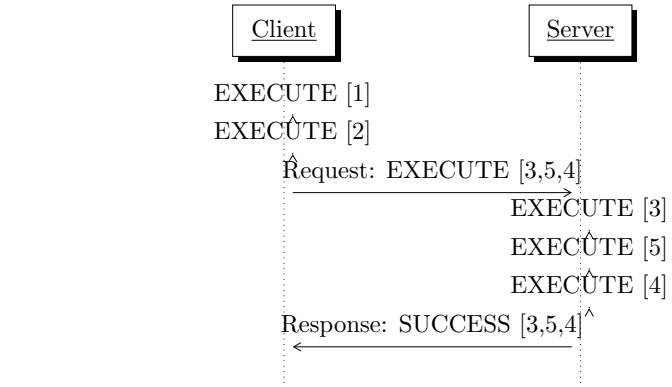
Many existing software systems solve a subset of the previous problems but do not address the entire range of issues.

1.4 Novel and Technical Contributions

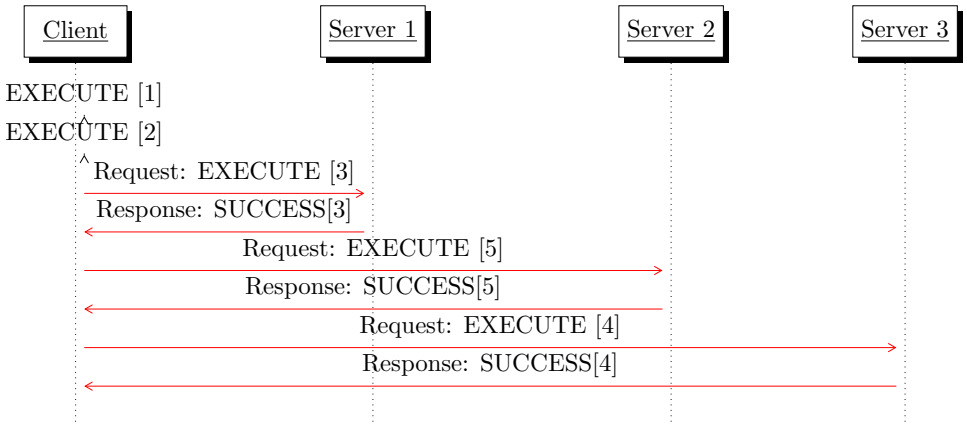
In this thesis, we present Distributed Cognitive Resource Allocation (DCRA), a novel end-to-end Gaming as a Service (GaaS) model, designed as a ubiquitous game platform solution. Unlike previous solutions, DCRA is Scalable, Available, Fault tolerant, and programmable model. DCRA leverages the state-of-the-art distributed programming techniques to achieve these goals.

In particular, *the novelty of DCRA is in how it uses traditional distributed hash tables, which are used typically as storage solutions, to distribute computation across many nodes. The properties of DHT solutions, such as consistent hashing, allow us to store and recover computational state easily, and this enables seamless mapping of stateful computation onto a distributed system with many compute nodes.*

Our evaluation using a wide collection of heavy network bandwidth, failsafe, and QoE tests shows that DCRA meets the expected design goals and has clear advantages over CRA models, especially when the cloud computation, bandwidth cost, QoE, and programmability requirements are important. Moreover, DCRA imposes additional effort on the client considering the diversity of its hardware specification, thus reducing the cloud overall cost. This makes DCRA very suitable for large scale gaming extensive computational environment that combine client flexibility and cloud servers computational power.



(a) Normal components execution (CRA).



(b) Dynamic components execution (DCRA).

Figure 1.1: Execution of Game Components.

Chapter 2

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

2.1 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 [20]. Figure 2.1 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.

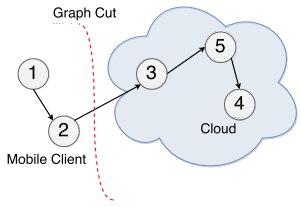


Figure 2.1: Game Components Partitioning.

2.2 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 total number of server nodes. We have tested our system on a 95 nodes on PlanetLab [21]. 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.

Figure 2.2 illustrates the naming of the server nodes in the overlay. For the game shown in Figure 2.1, 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.

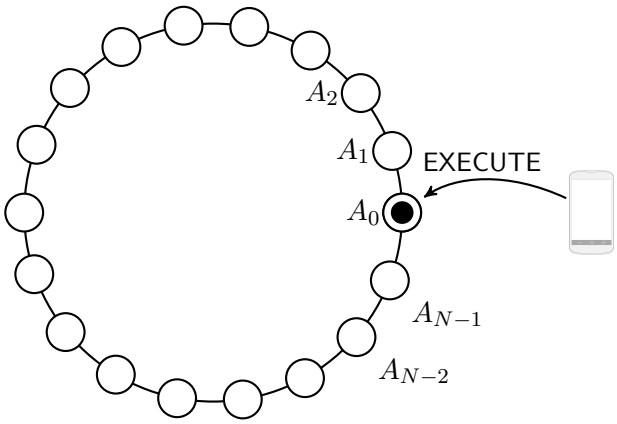


Figure 2.2: DCRA as an Overlay Network.

2.3 Membership Protocol

A Gossip-based membership protocol is implemented to obtain a global view of the state of each node in the system [22]. 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.

```
1 repeat
2 |   count = 0;
3 |   repeat
4 |   |   count ++;
5 |   |   send message to random node;
6 |   until count == log(N) nodes;
7 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:

$$time.time() - t\_last\_updated < T\_Fail \tag{2.1}$$

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. 2.2 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.

$$\begin{gathered}[remote\_HEARTBEAT > local\_HEARTBEAT] \\ AND \\ [(time.time() - t\_last\_updated < T\_Fail \\ OR \\ T\_Clean < time.time() - t\_last\_updated)],\end{gathered} \tag{2.2}$$

where T_Clean is defined as:

$$T\_Clean = 2 \times T\_Fail. \tag{2.3}$$

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.

2.4 Routing/Naming

Routing of a game's component is achieved by consistent hashing [23]. A *key* is defined as the hash of the concatenation of $GameID$ and $ComponentID$. Each *key* is assigned a server node by

$$hash(key)\%N \tag{2.4}$$

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 *key*s are evenly distributed across the system, some popular games might translate into highly executed *key*s. We leave this problem as future work, however, it can be seen from the Eq. 2.4 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) [24]. 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.

To our knowledge, we are the first to use DHT-based solution for fast routing of remote executions. While DHT-based protocols traditionally used to store and retrieve data, we used it to quickly execute and retrieve the output of a remote execution. The traditional routing schemes (i.e. broadcast and multicast) have already proved in the literature to set behind in terms of speed and efficiency.

2.5 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. Figure 2.3 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 Figure 2.1, 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 min-

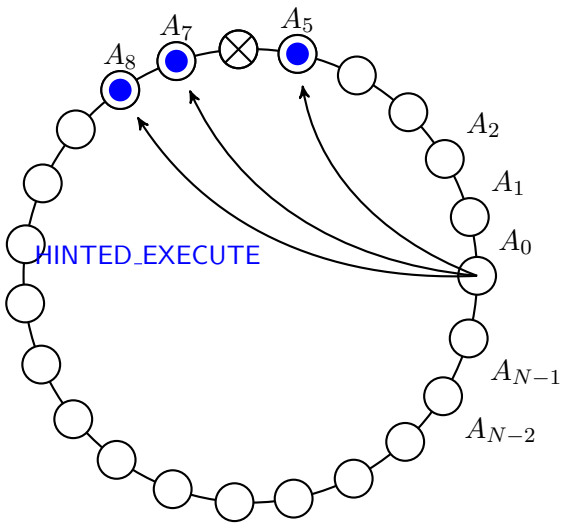


Figure 2.3: Process Resilience: Node A_0 sends R_f EXECUTE requests.

imize 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.

Chapter 3

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*.

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.

3.1 Fault Tolerance

Although hardware failures probability is too low, it is still a major factor in distributed system design in a large-scale systems [25]. 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).

3.2 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.

3.3 Performance Evaluation

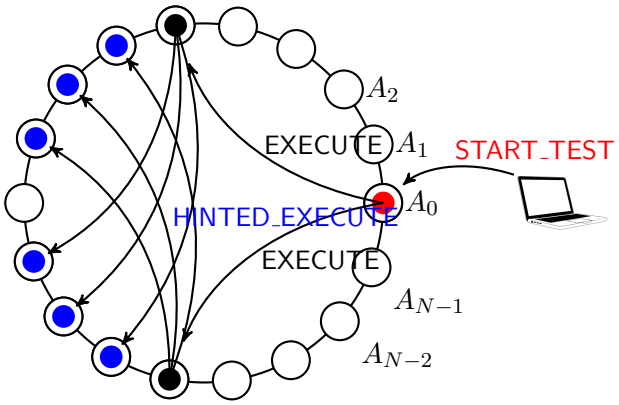
This objective of the test is to measure performance on large scale deployment. The test reports the response time and throughput of the system with low, medium, and high loads. In this test, an observer node sends

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:

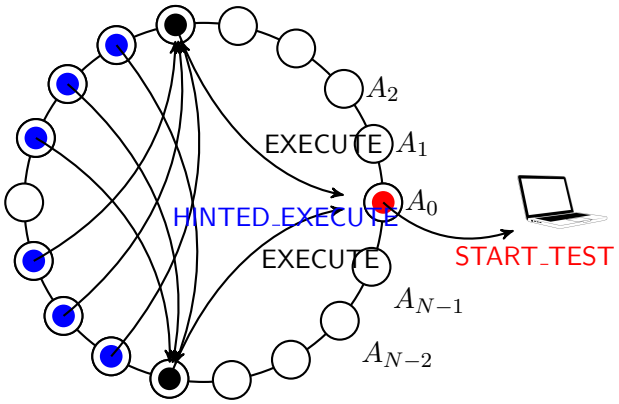
3.3.1 Multiple Nodes Test

In this variant, the EXECUTE commands are sent to other random nodes in the system as shown in Figure 3.1. Figure 3.2 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 Figure 3.2, 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 [26]. 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.



(a) Requests.



(b) Replies.

Figure 3.1: Multiple nodes test.

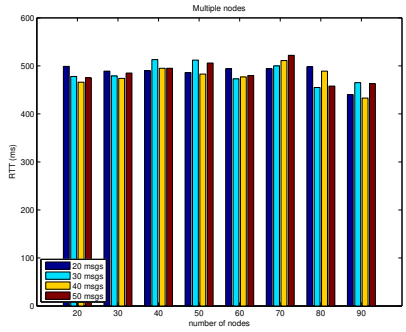
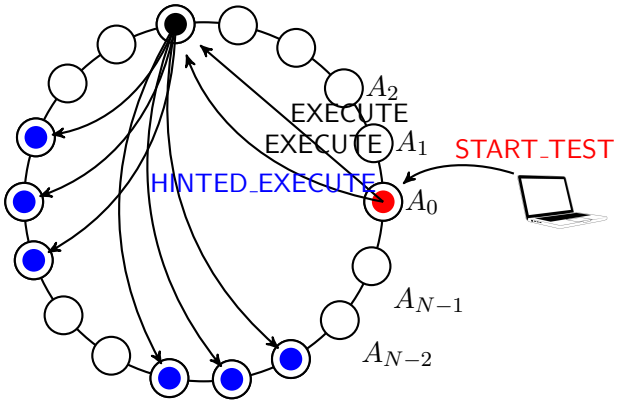


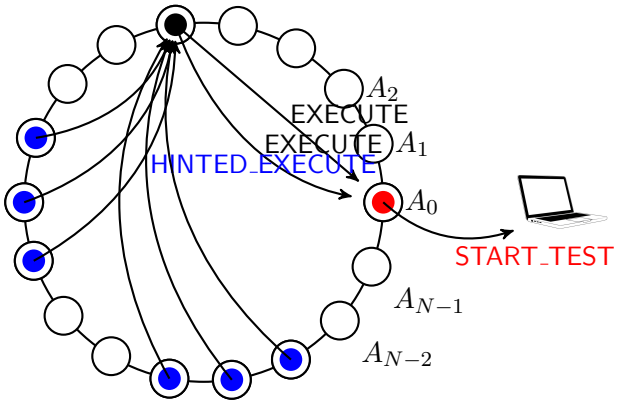
Figure 3.2: RTT for Multiple Nodes Test.

3.3.2 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 Figure 3.3, Multiple server nodes simulates the client nodes and send EXECUTE requests to the single node. It is shown in Figure 3.4 that the performance is worse than the multiple nodes test (Figure 3.2) 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.

Figure 3.3: Single Node Test.

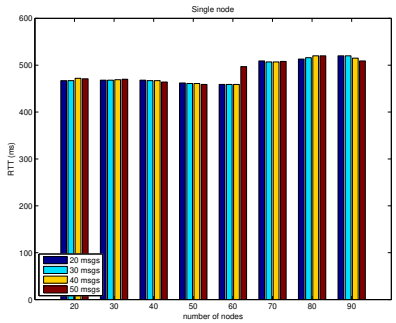


Figure 3.4: RTT for Single Node Test.

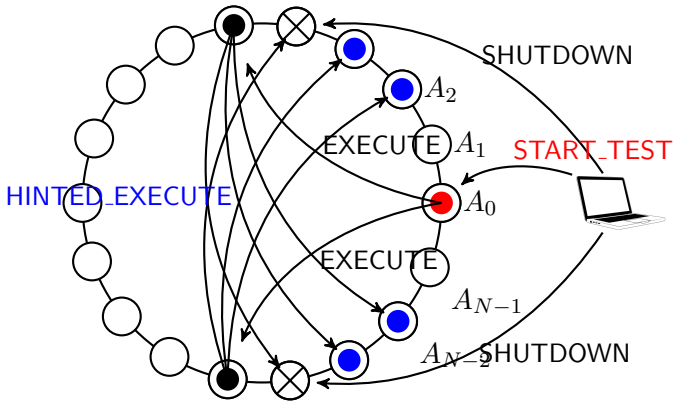
3.4 Catastrophic Failure Test

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. Figure 3.5 illustrates the scheme of the test. As shown in Figure 3.6, the RTT decreased after the failure, however, DCRA still function correctly. Hence, proofing the high convergence of its membership protocol. The system cover 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.

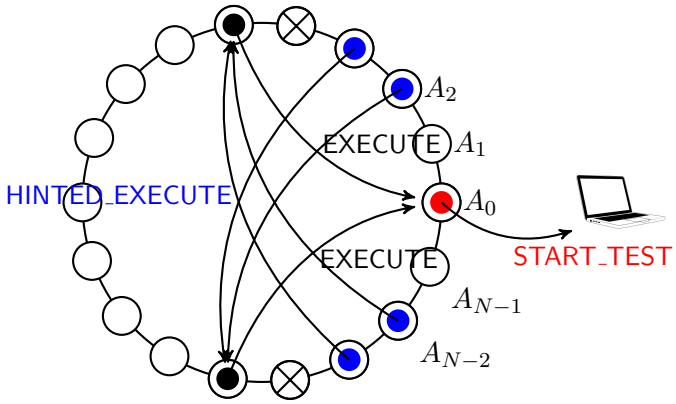
3.5 Client Design

A common problem in managing resources in cloud computing is resource allocation, where the problem is, how much resources should be reserved for a process if the demand is variable at any time. Figure 3.7 shows the possible solutions to this problem [27]. If the reserved resources are higher than the demand there will be waste of the resources. On the other hand, if the reserved resources are less than the demand, this might pause or halt the game execution. Dynamic provisioning is a robust solution where it is foreknown how much demand the client needs at any specific time. Hence, proper resources will be reserved.

From client perspective, we try to find the best approach to split the game's components between the client and cloud. Due to the unpredictability of network delay, the best split should be learned experience. Cognitive



(a) Requests.



(b) Replies.

Figure 3.5: Catastrophic Failure Test.

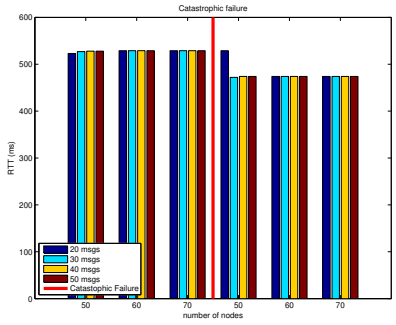
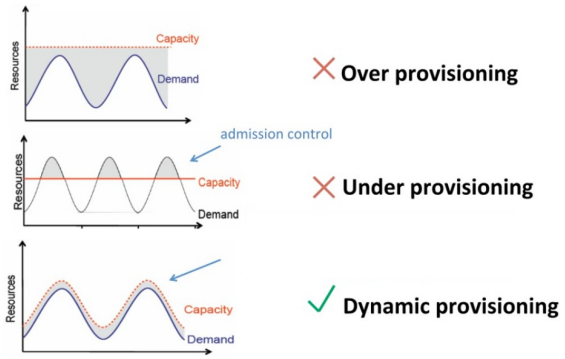


Figure 3.6: RTT for Catastrophic Failure Test.

Figure 3.7: Dynamic Resource Allocation.



model should adapt to the variation of terminal's resources and network dynamics. The fact that network delay environment is not deterministic, requires cognitive model that operate over by learning.

Learning can be categorized to supervised, unsupervised, and reinforced. In supervised learning a feedback signal is required from domain expert. Like supervised, reinforced agent this feedback is represented as a reward, not as desired output or value. The reward help to generalize the client model to adapt to different gaming setup environments such as processing power, available memory, and network conditions.

In our cognitive model a *partitioning coordinator* is the core element where the game's components is redirected to either the cloud or the terminal. This requires applying mathematical models that translates high level performance [14]. However, these models are sensitive to the time-varying application's demand which exists in cloud gaming. We applied reinforcement learning as a dynamic provisioning solution to the resource allocation problem which saves a considerable amount of the terminal and cloud resources since the partitioning coordinator decision is adaptable to the variation of the terminal's network and the cloud dynamics over time. For example, sudden changes in the terminal's network dynamics might cause the network delay to increase and hence, it is not guaranteed that the terminal will be able to host a game's component it used to be able to host in the past. Such a problem requires a partitioning coordinator model in which the decision can be made on the fly.

This client model will provide the cognitive model with the following properties:

- Partitioning on the fly: There is no need to train the cognitive model

to adapt to the new network environment changes.

- Unified: There is no need train the *partitioning coordinator* for all variations of network environment. The model will be able to find a good trade off for partitioning the game component between the terminal and the cloud to maximize the performance. Hence, this client model is providing provisioning of quality assurance.
- Autonomous: No thresholds need to be set. This property allows the model to generalize better on the long run.
- Adaptive: Learn from experience (trial and error). Hence, the training can re-run without any additional programming [33].

3.5.1 System Architecture

We train the cognitive model by using reinforcement learning to make the decision where to execute a game component by learning from experience. The general goal is to maximize the throughput and minimize the delay. To demonstrate the model, a set of state-action is defined with initially a random probability value reflecting the probability of each action (A_n) for each state (S_n). We may define the states as:

- S_1: Set of components allocated in the cloud
- S_2: set of components allocated in the terminal
- S_3: The available resources for the terminal
- S_4: The available resources for the cloud

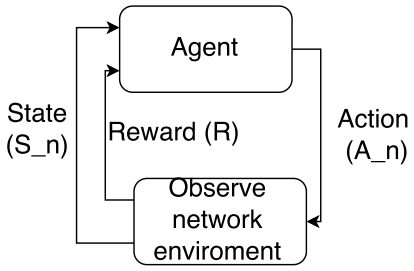
and the actions:

- A_1: Allocate component C_n on the cloud
- A_2: Allocate component C_n on the terminal

For any state, there is an optimal action that has a maximum probability $Q(S_n, A_n)$ which is also referred to as an action-value or Q-function [34]. Figure 3.8 shows the scheme of the model. After an action is executed a reward and a new state are observed. The reward is where the experience

is being taken into account. If a positive reward is recorded (i.e. latency decreased), then the action that has been just performed is a good action, hence, its probability will increase. On the other hand, if a negative reward is observed (i.e. bandwidth decreased), then the action is a bad action, hence, its probability will decrease. Over time, the probabilities for all the state-action pairs has been proved to converge [35]. Hence, a proper action or placement of a game component will be set for each state of the environment.

Figure 3.8: System Architecture (Partitioning Coordinator).



The longer the model runs, the better actions and performance will be, since the agent is learning from its mistakes. Table 3.1 shows how the look up table (LUT) for all the state-actions will be stored in the memory of the cognitive engine on the client side. The output of the model is defined as which action yield the best reward. Although the size of the LUT table can get very large because all the combinations of states and the actions (curse of dimensionality), we have applied some reprocessing techniques to make as small and efficient as possible. One of these techniques, is dimenstionality reduction where the range of delay is partitioned to finite set of a fixed size. Although, preprocessing will cause loss of information, it will ease the learning speed and significantly improves generalization.

The algorithm for the Agent to calculate the $Q(S_n, A_n)$ probabilities: where α is a learning rate constant (0-1) typically close to 0 to control the speed of learning and γ is a discount factor constant (0-1) which is typically close to 1 to weigh the future rewards. S' is the state of client after action [36].

Table 3.1: Lookup Table (LUT) for DCRA Cognitive Model.

S_1	S_2	S_3	S_4	Action	Probability ($Q(S_n, A_n)$)
30	2	340	104	A_1	.4
30	2	340	104	A_2	.3
..	..	..	..	..	..

Initialize $Q(S_n, A_n)$;
repeat
  Initialize S_n;
  repeat
    Choose A_n from S_n (maximum);
    perform action A_n, record reward (R) and future state (S'_n);
    $Q(S_n, A_n) =$
     $Q(S_n, A_n) + \alpha[R + \gamma \times max(Q(S'_n, A'_n)) - Q(S_n, A_n)]$;
    $S_n = S'_n$;
  until;
until;

Algorithm 2: Partitioning algorithm

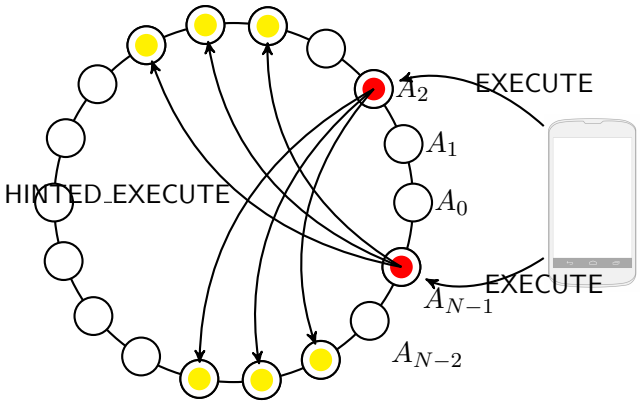
3.5.2 Assumptions

- Finite number of states: The agent is Markov Decision Process (MDP) where there is finite number of states which yield to a problem with generalization. However, many techniques can be performed to solve this problem such as using approximation function using neural networks to have continuous range of state values. This is left as future work.

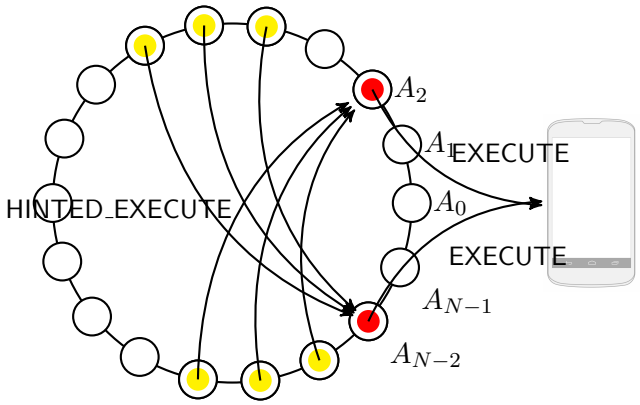
- The best action is always the action with the highest probability (greedy): This is not always the best solution in real world problems [36]. For example, the agent may choose a sequence of actions which yield to bad rewards eventually. This is a well known problem that can be solved by exploration from time to time by choosing a random action.

3.6 Client Evaluation

Python client simulator were deployed on PlanetLab and we ran the Partitioning Coordinator with all the components initially running locally. As the agent starts learning, it will start offload game components, record the reward, and observe the new state. Figure 3.9 illustrates a client setup. The server list is provided for the client initially so it would offload components to the cloud servers as discussed in Section 2.4. Figure 3.10 illustrates the convergence of client partitioning for different parameters. It is shown that the cognitive model has succeeded to convergence as time passes.

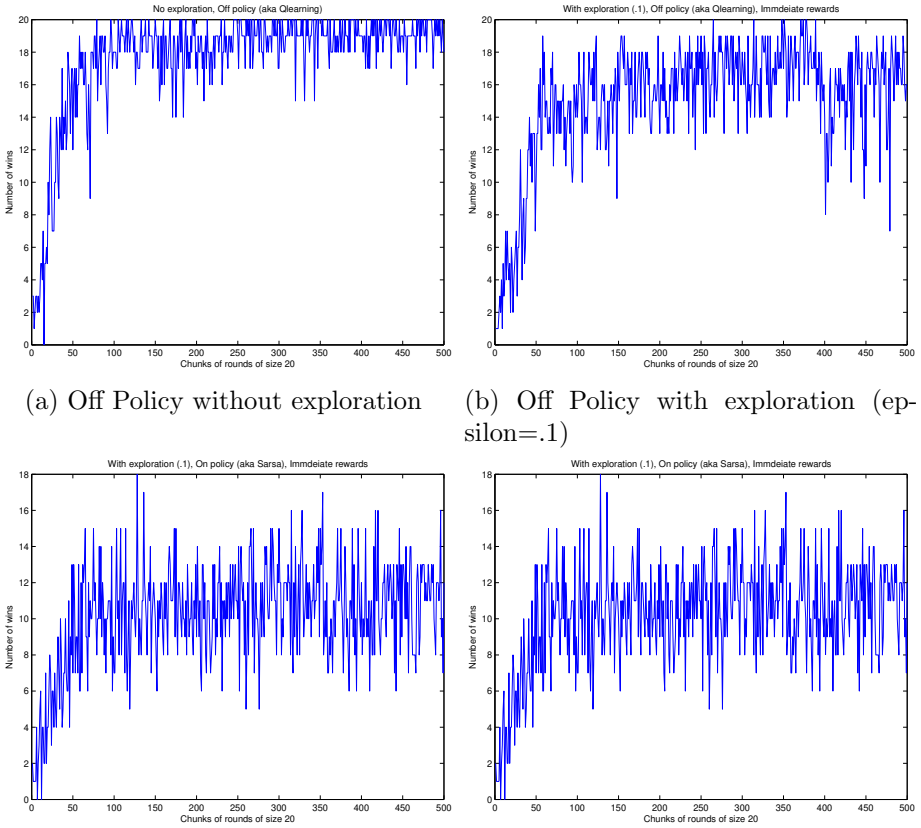


(a) Requests.



(b) Replies.

Figure 3.9: Partitioning Coordinator.



(a) Off Policy without exploration

(b) Off Policy with exploration (epsilon=.1)

(c) On Policy without exploration

(d) On Policy with exploration (epsilon=.1)

Figure 3.10: Progress of Learning and Convergence of Learning Measures.

Chapter 4

DCRA: Middleware

Distributed Mobile gaming involves thousands of entities whose location and behavior might vary thought out the lifetime of the system. The need for building abstractions to build a distributed application for such a heterogeneous environment at multiple scales is an urging research area for the next decade [27]. Currently, for each application, a full stack of services needed to be implemented from scratch. Such services might be a key-value store, coordination protocol, membership protocol, distributed objects, consistency guarantees, and state synchronization. Developing a middleware that works in changing and the mobile environment is a challenge. These constraints force researches to develop new methods of coordination to be more dynamic and decoupled. In such an environment, the middleware has to automate and scale these services. Also, the middleware has to facilitate services reuse across multiple applications domains. However, such automation may intensify complexity and affects applications correctness. Hence, substantial proof testing is needed for such middleware. New middleware is needed to be designed to reduce the burden on application designers. In such large scale settings, this middleware will glue different services to provide an adequate communication scheme.

4.1 Network Microservices

We use network microservices that consist of a collection of loosely-coupled, independent, and deployable services to construct the middleware. These services work together to store past data and meditate communication of future data. The network microservices can be installed to run on a cluster on the edge by the client premise or in centric-cloud.

We implemented a quorum based consensus algorithm to provide both SC and EC to read/write operations based on algorithms described on [28–30]. Quorum-based consistency protocols have flexibility in term of tuning the strength of consistency among multiple replicas. In such a protocol, each *object* is replicated in a quorum consists of N nodes. Each object is associated with a vector stamp corresponds to the last modified logical time.

The larger the stamp is, the most recent the state of the object is. Write operation needed to collect the approval of W nodes out of N. Similarly, a read operation needs R votes out of N. $W + R$ has to be more than N to ensure read and write quorums intersect. Hence, a read operation will not retrieve the stale state of an object. Also, W has to be larger than $N/2$. Hence, no conflicts in write operations can occur. These two conditions are satisfactory to the correctness of the protocol. Changing the settings of W and R can provide different levels of consistency [29]. For example, setting R to N can ensure that a read operation will surely see the last write. Hence, strong consistency is guaranteed.

It is worth to note that the protocol does not provide any synchronization. To force causality, developers have to use our provided distributed locks. This behavior mimics distributed mutual exclusion and hence provide a close effect to linearizability.

4.2 Publish/Subscribe (Pub/Sub) Messaging

Pub/sub messaging is used as a form of an asynchronous reliable communication channel between client instances [31]. In such a model, subscribers register their interest in a channel and subsequently, receive a notification generated by publishers.

We avoided other means of communication. Namely, distributed shared memory (DSM) model, message passing, and remoter procedure calls (RPCs). They are not decoupled in time, space, and synchronization. Client or server do not need to know about each other physical addresses. Instead, logical addressing can be set up through the developer form the code base. Also, the client and server do not have to communicate actively at the time to exchange information. Instead, messages can spread out lazily. Lastly, servers and clients should not be blocked while receiving and send information, respectively. Decoupling space, time, synchronization, increase the scalability of applications. Hence, we felt that the Pub/Sub pattern fits as a mean of communication for future data.

4.3 API and Language

To mediate the interaction between the developer's intentions and the middleware services, we have designed an API library to reside on the client side. It also abstracts the connection between the middleware language and network microservices by only exposing objects or actions the developer needs.

The API mainly consists of:

- Thin Client: To attend the queue of outgoing requests passed by the Interpreter.
- Thin Server: To attend the queue of incoming requests coming from the network microservices cluster.
- Interpreter: To execute primitives and pass requests to the queue of outgoing request in an asynchronous manner.

We have also designed a high-level language based on Lox language [32] tailored specifically to communicate with middleware's network microservices. The interpreter hammers user's program into a form that network microservices can understand. In this subsection, we discuss concepts, and design choices while writing the interpreter to meet the demands of the proposed system.

We chose Python as an implementation language basically because it is a widely used object-oriented language. Unlike scripting languages, less functionality is hidden while coding. Also, object-oriented languages are still the dominant way of organizing and encapsulating data. The source code of the middleware language is similar to any other high-level language. More like C family. It is summarized in Figures .1, .2, .3, and .4.

A program in this language is a series of declarations (Figure .1), which are the statements that bind new identifiers or any of the other statement types. Remaining statement rules produce side effects like reading input or producing output but do not introduce bindings (Figure .2). The rules of expressions that produce values are shown in Figure .3. Rules that a developer needs to interact with the middleware's network microservices are shown in Figure .4.

The interpreting process is consisting mainly of a Scanner, Parser, and an Interpreter. These three components construct a pipeline in which middleware network *commands* is produced from source code. The first step is to scan (lex analysis) the source code to convert it to a sequence of Tokens. Possible tokens values examples are a character, number, a string literal, or identifiers. Parser constructs syntax trees from a flat sequence of tokens generated in the previous step based on the grammar rules mentioned earlier. Basic data types: Booleans, Numbers, Strings are supported. Various expressions are supported: Arithmetic, Comparison and equality, and Logical operators with the support of Precedence and grouping. Control Flow and Functions are also supported. Classes and advanced data structures are left for future work.

For simplicity, the language is dynamically typed, no type checking is done in compile time. Instead, it will be reserved for runtime. Also, we left automatic memory management through reference counting or garbage collection for future work. All the semantic insights are kept stored in and a *environment* data structure to support variables binding and declarations and to add relations to nested scopes.

Although the interpreter has a syntax error report mechanism, a user is responsible for making sure that the written code if free of errors. The interpreter takes the source code to execute it immediately. The interpreter can access local and global user's code variables. Illustrated by the *python_ref* production rule in Figure .3. Hence, the language code can be written seamlessly between the application logic components without the need to deal with any external interfaces.

Chapter 5

Conclusions

We have implemented a decentralized cognitive resource allocation system to host a platform to run mobile games over the cloud. The system matches the required objectives: availability, fault tolerance, and programmability 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 system's performance evaluation shows that DCRA succeeds CRA in terms of throughput and RTT.

To our knowledge, we are the first to use DHT-based key-value stores as a routing technique and as a process resilience abstract which led DCRA to satisfy the required delay objectives. Other approaches used in the literature proved to slow up the total performance.

We, have also cover the heterogeneity of clients in edge computing architecture through the separation of the logic of the application and needed network microservices: data store, request-response protocol, consistency, mutual exclusion and caching support. 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.

Chapter 6

Future Work

Several challenges face DCRA and need to be addressed in order to make it practical solution to Mobile gaming. We plan to address these challenges later as a future work.

6.1 DCRA Bandwidth Cost

Problem: *Minimize system generated traffic overheads without compromising user gaming experience.*
Several studies has proven the significant amount of redundant traffic in distributed systems generated traffic [5, 6]. Bandwidth cost, in contrary to QoE, is not a tangible property to the player. Hence, It is unclear how much mobile games generated redundant and unnecessary traffic that overcharge the user data plan. Although a good QoE require the game to send frequent updates, we try to find the balance point in which the player is not being speed lowered or being overcharged because of the high volume updates neither frustrated by its low updates.

Many techniques has been proposed in the literature to decrease bandwidth cost. Authors in [37], designed learning model to detect and minimize the cost by training using the sent TCP traffic patterns. Although the model succeeded to capture short and long term data redundancy, the model does not detect different levels of redundancy such as cloud-cloud redundant communication. In [38], authors build a software cache in receiver TCP code in which all packets are hashed for future detection of redundant traffic. This is similar to DCRA cache in the sense of avoiding sending multiple replies for the same request.

Throttling is the current business model for mobile data plans which is replacing flat-rate plans [39]. In throttling, full bandwidth service is provided to the user until a certain threshold is reached. Once the threshold is reached, a throttled bandwidth is provided to the user until the end of the payment cycle. Hence, the study of the generated traffic assure no redundant or unnecessary packets are being sent. DCRA provides a middleware layer to break the dependency between a game implementation and the gen-

erated traffic. Hence, game data can be generated, collected, and studied to understand the problem.

Since sending redundant and unnecessary gaming traffic may overcharge the players, DCRA aims to prevent such overcharges by sending only necessary data that aggregate the information of multiple packets. Further study of decreasing the "repair bandwidth" where a server while restarting from a crash needs information (Bandwidth) from other peers about its state [40]. is also needed. Small GaaS design decisions affect bandwidth cost heavily. For example, games usually transmit repeated packets to counter loss and delayed packets. DCRA overcome this problem by applying several techniques such as using UDP over TCP and using software caches at several communication layers.

We plan to study the generated traffic out of DCRA and relate its factors to the cost. Later we will build a model to predict the cost on the go. We intend to apply machine learning prediction and classifying algorithms to aid this purpose.

6.2 DCRA Energy Consumption Cost

Problem: *Optimize game energy Consumption to increase mobile device operational time without compromising user gaming experience.*

One of the most important factor that led mobile devices to be the most dominant mean of communicating is its efficient energy [41]. It has been studied that battery time is the most important factor to select a new mobile device while be compared to storage, camera, or any other mobile device specifications [42]. Advanced video games are computational expensive. Hence, battery drains very fast while playing video games in comparison to other mobile applications. The only solution to this problem is either to extend the life time of mobile battery or to utilize resources in a smarter way [43]. We argue that DCRA save mobile energy better than other GaaS systems for the same environment settings by including the terminal processing and memory usage information as part of the agent state in DCRA *partitioning coordinator*.

Literature papers have a thorough studies of how to reduce energy consumption cost in hardware, communications, protocols, and recently in Internet applications. Authors in [44] studied the trade-off between energy cost and QoS. They emphasis the role of routing protocol and load balancing of achieving such healthy state of what they call self ware network (SAN). The finding affirms the need of good routing and load balancing techniques such

ones as implemented in DCRA and assert the need of further study of how to enhance these features. In [45], the authors study the need to apply energy cost reduction in all of the OSI model including the application layer. Also, the user energy settings and selection history should be taken into the account since the business environment of the cloud system might influence these settings. Authors in [46] emphasis the importance of having a soft state attached to each server in the cloud where an inefficient energy server will delegate processing to an efficient energy server. Energy efficient server is for example a device that is always ON.

In DCRA, We study energy consumption in terms of signaling, routing, load balancing , and having soft states. We also plan examine the business environment work flow of the user to balance QoE with energy saving.

6.3 DCRA Gaming Quality Experience

Problem: *Analyzing the factors that affect gaming experience for DCRA and measure QoE on the fly.*

The interactive nature of mobile gaming and the fact that network conditions are not perfect, led to the unexpected high response time while playing. The importance of having better QoE for the user is to eliminate the barrier between mobile gaming and console gaming. It is a vital information for game designer so the game design can be enhanced. To understand the factors that affect QoE, we study the client and cloud delays, the wireless network delays, game genre, and game settings. To measure QoE, we examine the subjective rating of QoE for various variations of the previous factors [47]. Later, we will build a model to measure the current gaming experience based on the data we collect for the subjective test.

Bibliography

[1] C. Feijoo, J. Gómez-Barroso, J. Aguado, S. Ramos, "Mobile gaming: Industry challenges and policy implications," *Telecommunications Policy*, vol. 36, no. 3, 2012, pp. 212–221.

[2] N. Al-Rousan, W. Cai, H. Ji and V. Leung, "DCRA: Decentralized Cognitive Resource Allocation Model for Game as a Service", *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[3] D. Mishra, M. El Zarki, A. Erbad, C. Hsu and N. Venkatasubramanian, "Clouds + Games: A Multifaceted Approach", IEEE Internet Computing, vol. 18, no. 3, pp. 20-27, 2014.

[4] M. Claypool and K. Claypool, "Latency can kill", *Proceedings of the first annual ACM SIGMM conference on Multimedia systems - MMSys '10*, 2010.

[5] N. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic", *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 87-95, 2000.

[6] A. Anand, C. Muthukrishnan, A. Akella and R. Ramjee, "Redundancy in network traffic", *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems - SIGMETRICS '09*, 2009.

[7] F. Kong and X. Liu, "A Survey on Green-Energy-Aware Power Management for Datacenters", *ACM Computing Surveys*, vol. 47, no. 2, pp. 1-38, 2014.

[8] N. Betty, "Scale in Distributed Systems", *Readings in Distributed Computing Systems*, Los Alamitos, CA: IEEE Computer Society Press, pp. 463-489, 1994.

[9] P. Veríssimo and L. Rodrigues, Distributed Systems for System Architects, 1st ed. NewYork: Springer-Verlag New York Inc, 2013, pp. 44-50.

[10] 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.

[11] OnLive [Online]. Available: http://onlive.com/

[12] Gaikai [Online]. Available: https://www.gaikai.com/

[13] S. Chuah, N. Cheung and C. Yuen, "Layered Coding for Mobile Cloud Gaming Using Scalable Blinn-Phong Lighting", *IEEE Transactions on Image Processing*, vol. 25, no. 7, pp. 3112-3125, 2016.

[14] 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.

[15] R. Shea, Jiangchuan Liu, E. Ngai and Yong Cui, "Cloud gaming: architecture and performance", *IEEE Network*, vol. 27, no. 4, pp. 16-21, 2013.

[16] W. Cai, Z. Hong, X. Wang, H. Chan and V. Leung, "Quality-of-Experience Optimization for a Cloud Gaming System With Ad Hoc Cloudlet Assistance", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, pp. 2092-2104, 2015.

[17] M. Amiri, H. Osman, S. Shirmohammadi and M. Abdallah, "Toward Delay-Efficient Game-Aware Data Centers for Cloud Gaming", *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 12, no. 5, pp. 1-19, 2016.

[18] Y. Lin and H. Shen, "Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Experience", *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015.

[19] Y. Lin and H. Shen, "Cloud Fog: Towards High Quality of Experience in Cloud Gaming", 2015 44th International Conference on Parallel Processing, 2015.

[20] 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

[21] PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services [Online]. Available: https://www.planet-lab.org

[22] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Service,T Proc. Conf. Middleware*, pages 55–70, 1998.

[23] 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.

[24] A. Tanenbaum, M. V. Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007

[25] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote from LADIS, 2009.

[26] IP Latency Statistics [Online]. Available: http://www.verizonenterprise.com/about/network/latency/

[27] 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.

[28] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994

[29] Vogels, Werner , "All Things Distributed", Werner Vogel's weblog on building scalable and robust distributed systems; http://www.allthingsdistributed.com/2009/05/amazon-cloudwatch.html; May 17, 2009, pp. 1-3.

[30] H. Attiya, A. Bar-Noy, and D. Dolev. *Sharing memory robustly in message-passing systems*. Journal of the ACM, 42(1):124-142, Jan. 1995

[31] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *computing surveys (CSUR).*

[32] R. Nystrom, 'Crafting Inerpreters', 2018. [Online]. Available: http://www.craftinginterpreters.com/introduction.html. [Accessed: 10- Jan- 2019].

[33] 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.

[34] C. J. C. H. Watkins, "Learning with delayed rewards," Ph.D. dissertation, Psychology Dept., Univ. of Cambridge, UK, 1989

[35] C. Watkins and P. Dayan, "Q-learning", Machine Learning, vol. 8, no. 3-4, pp. 279-292, 1992.

[36] R. Sutton and A. Barto, *"Reinforcement Learning"*. MIT press, 1998.

[37] L. Yu, H. Shen, K. Sapra, L. Ye and Z. Cai, "CoRE: Cooperative End-to-End Traffic Redundancy Elimination for Reducing Cloud Bandwidth Cost", *IEEE Transactions on Parallel and Distributed Systems*, pp. 12-18, 2016.

[38] E. Zohar, I. Cidon and O. Mokryn, "PACK: Prediction-Based Cloud Bandwidth and Cost Reduction System", *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 39-51, 2014.

[39] D. Chen, P. Lin, K. Chen, "Does Playing Online Mobile Games Overcharge You for the Fun?", in *Proceedings of the 2Nd Workshop on Mobile Gaming* (New York, NY, USA, May 04 - 06, 1997). MobiGames '15. ACM Press, New York, NY, pp. 13-18.

[40] S. Akhlaghi, A. Kiani and M. Ghanavati, "Cost-bandwidth tradeoff in distributed storage systems", *Computer Communications*, vol. 33, no. 17, pp. 2105-2115, 2010.

[41] S. Chuah, C. Yuen and N. Cheung, "Cloud gaming: a green solution to massive multiplayer online games," in *IEEE Wireless Communications*, vol. 21, no. 4, August 2014, pp. 78-87.

[42] L. Liu, H. Wang, X. Liu, X. Jin, W. Bo, Q. Bo, and Y. Chen, "GreenCloud: A New Architecture for Green Data Center", in *Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session* (Barcelona, Spain, May 04 - 06, 2009). ICAC-INDST '09, pp. 29-38.

[43] Question: Energy Use of Internet", Sarokin, D., 2017. [Online]. Available: http://uclue.com/?xq=724. [Accessed: 2- Jan- 2017].

[44] E. Gelenbe, "Steps toward self-aware networks", *Communications of the ACM*, vol. 52, no. 7, p. 66, 2009.

[45] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Dang and K. Pentikousis, "Energy-Efficient Cloud Computing", The Computer Journal, vol. 53, no. 7, pp. 1045-1051, 2009.

[46] S. Ba, E. Oki and I. Ouédraogo, "Reducing the power consumption of hose-model networks with bundled links", *IET Networks*, vol. 4, no. 2, pp. 119-127, 2015.

[47] S. Wang, S. Dey, "Cloud Mobile Gaming: Modeling and Measuring User Experience in Mobile Wireless Networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 16, no. 1, Jan. 2012, pp. 10–21.

[48] 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.

[49] 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.

DCRA Middleware's Language Syntax

Program .1 Declarations Rules.

```
program     -> declaration* EOF ;
declaration -> funDecl
             | varDecl
             | statement ;

funDecl -> "fun" function ;
varDecl -> "var" IDENTIFIER ( "=" expression | read )? ";" ;
```

Program .2 Grammar Rules.

```
statement  -> exprStmt
           | forStmt
           | ifStmt
           | printStmt
           | returnStmt
           | whileStmt
           | block
           | loteosCommandStmt
           ;
 exprStmt  -> expression ";" ;
 forStmt   -> "for" "(" ( varDecl | exprStmt | ";" )
                        expression? ";"
                        expression? ")" statement ;
 ifStmt    -> "if" "(" expression ")" statement ( "else" statement )? ;
 printStmt -> "print" expression ";" ;
 returnStmt -> "return" expression? ";" ;
 whileStmt -> "while" "(" expression ")" statement ;
 block     -> "{" declaration* "}" ;
```

Program .3 Grammar Helper Rules.

```
expression -> assignment ;
assignment -> python_ref | IDENTIFIER "=" assignment | read
           | logic_or ;
logic_or   -> logic_and ( "or" logic_and )* ;
logic_and  -> equality ( "and" equality )* ;

equality       -> comparison ( ( "!=" | "==" ) comparison )* ;
comparison     -> addition ( ( ">" | ">=" | "<" | "<=" ) addition )* ;
addition       -> multiplication ( ( "-" | "+" ) multiplication )* ;
multiplication -> unary ( ( "/" | "*" ) unary )* ;

unary -> ( "!" | "-" ) unary | call ;
call  -> primary ( "(" arguments? ")" )* ;
primary        ->  NUMBER | STRING | "false" | "true"
               | "nil"
               | "(" expression ")"  | python_ref
               | python_ref | IDENTIFIER ;
python_ref     -> "@" IDENTIFIER
function -> IDENTIFIER "(" parameters? ")" block ;
parameters -> IDENTIFIER ( "," IDENTIFIER )* ;
arguments -> expression ( "," expression )* ;
```

Program .4 Middleware Related Grammar Rules.

```
loteosCommandStmt -> loteosCommand ";" ;
loteosCommand -> assert | command

assert -> "assert" "(" command, consistency_level ")" ;
command -> read | write | remove | lock | unlock |
          register | subscribe | publish

read -> READ "(" primary ")" ;
write -> WRITE "(" primary "," expression")" ;
remove -> REMOVE "(" primary ")" ;
lock -> LOCK "(" primary ")" ;
unlock -> UNLOCK "(" primary ")" ;
register -> REGISTER "(" primary ")" ;
subscribe -> SUBSCRIBE "(" primary ")" ;
publish -> PUBLISH "(" primary ")" ;

consistency_level -> SC | EC | MC ;
```