MEERKAT: EXTENDING ENTITY-BASED PROGRAMMING TO NETWORKED GAMES

by

MD AMEER HAMZA

A thesis submitted to the School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
August 2012

Copyright © Md Ameer Hamza, 2012
Abstract

Game development is a complex and time-consuming activity that requires domain-specific knowledge and implementation skills. Networked games are particularly difficult due to the additional challenges of implementing the distribution. In recent years, game development has been simplified through tools that allow game development based on entities (objects that compose the game, e.g., avatar, vehicles, trees, and monsters). Entity-based tools simplify game programming by providing entity-level constructs and abstractions to the game developer. However, current entity-based tools fail to appropriately address the development of networked multiplayer games; they either do not support network gaming at all, or compromise the purity of the model by exposing low-level network programming to the game developer.

In this thesis, we present a pure entity-based model for developing networked multiplayer games. In our model, the game developer is completely shielded from network programming concerns. In order to demonstrate the model’s practicality, we implemented a game development toolkit called Meerkat. Meerkat uses a combination of generic distribution algorithms and a proxy-based architecture to provide a pure entity-based game programming interface. The same interface can be used to develop both distributed and non-distributed games. Meerkat automates all aspects of networking for the game developer. To evaluate the performance of our system, we built three multiplayer games of different genres. Our experiments show that the overhead of using fully-automated networking can be acceptable for a wide range of games, except in extreme cases where there are strict performance requirements. Meerkat demonstrates that it is possible to extend the pure entity-based approach to networked games while ensuring sufficient performance.
Acknowledgements

I wish to extend my most sincere thanks to my supervisor, Dr. Nick Graham, for the great care and individual guidance he provided throughout my research. The work in this thesis would not be possible without the exciting ideas and practical wisdom he shared every step of the way.

Thanks to all the current and past members of the EQUIS lab for making it such a wonderful place to work and have fun. Special thanks to Zi Ye, Cheryl Savery, Hamilton Hernandez, and Tad Stach for their continuous help, suggestions, and encouragement.

I could not be more thankful for the incredible support of my parents and all the friends I have been blessed with. Special thanks to Samee Zahur, who has been a great inspiration. My heart-felt thanks to my friends: Mohammad Hossain, Anton Barua, Umme Hunny, Shahed Zaman, Kazi Ahmed, Shajib Kumar, Hasanuzzaman (and many others) - who never let me feel that I was away from home.

Above all, I’m grateful to the almighty Allah for always taking me through all the hurdles and for bestowing me with His great bounties – through means I would never possibly have imagined.
# Table of Contents

Chapter 1 Introduction ................................................................. 1

Chapter 2 Background ................................................................. 8
  2.1 Basics of Game Development ............................................... 8
    2.1.1 Game Development Process and Participants .................. 8
  2.1.2 Game Development Tasks ............................................... 10
    2.1.2.1 General Tasks ......................................................... 10
    2.1.2.2 Distribution Programming Tasks ................................. 14
  2.2 Entity-based Game Development ......................................... 19
    2.2.1 Pure Entity-based Tools ................................................ 20
    2.2.2 Entity-based Tools for Networked Games ....................... 23
    2.2.3 Where Meerkat Fits ..................................................... 30
  2.3 Conclusion ........................................................................... 31

Chapter 3 The Meerkat Programming Model ................................. 32
  3.1 Overview of Meerkat ............................................................ 32
  3.2 Game Development with Meerkat ........................................ 33
    3.2.1 Snowballs – An Example Game ..................................... 34
    3.2.2 Entity Classes ............................................................. 35
    3.2.3 Entity Drawing and Animation ...................................... 35
    3.2.4 Entity State ............................................................... 37
    3.2.5 Entity Physics ............................................................ 38
    3.2.6 Input Response, Game-logic, AI .................................... 40
    3.2.7 Entity Interaction and Communication ......................... 43
    3.2.8 Bringing it Together .................................................... 45
    3.2.9 Making the Game Multiplayer ....................................... 49
  3.3 High-level Parameters for Controlling Distribution ................. 52
  3.4 Conclusion ........................................................................... 56

Chapter 4 Implementation ............................................................ 57
  4.1 Meerkat Model and Single-Player Engine ............................. 57
    4.1.1 The Game-loop and Key Elements ................................. 58
    4.1.2 Drawing and Animation .............................................. 60
    4.1.3 Adding Game Physics ................................................. 61
List of Figures

Figure 1: Screenshot of Need for Speed (a racing game) .............................................................. 12
Figure 2: Screenshot of Age of Empires (an RTS game) .............................................................. 13
Figure 3: Programming in Kodu using conditions and actions ...................................................... 21
Figure 4: Screenshot of the “Snowballs” game developed using Meerkat ................................. 34
Figure 5: Entity-hierarchy in the Snowballs game ......................................................................... 35
Figure 6: Entity members related to drawing and animation ........................................................ 36
Figure 7: Inherited and custom-defined state variables for Aircraft class ................................. 38
Figure 8: Core members of Entity for controlling physics behavior ........................................... 39
Figure 9: Custom scene-class named “SnowballsGame” ............................................................... 46
Figure 10: Core members of Entity and Scene classes ............................................................... 47
Figure 11: Two games using two different interest radius values ............................................... 54
Figure 12: Basic Meerkat game-loop ........................................................................................... 58
Figure 13: Modified Meerkat game-loop combining Client, Scene, and Entity classes ............... 59
Figure 14: Game rendering process in Meerkat .......................................................................... 61
Figure 15: Member mappings between Meerkat Entity and underlying physics objects ........... 63
Figure 16: Physics processing steps in Meerkat ......................................................................... 64
Figure 17: Input-handling integrated into the game-loop ................................................................. 65
Figure 18: Entity creation and spawning ..................................................................................... 66
Figure 19: Command structure for Meerkat server and clients .................................................. 68
Figure 20: Meerkat client-server communication architecture using Timelines .......................... 70
Figure 21: Entity replication architecture in Meerkat (shown for two clients) ......................... 73
Figure 22: Different clients own different entities for partitioning state and computation .......... 76
Figure 23: Interest calculation for a client that owns entities A1 and M1 ..................................... 79
Figure 24: Proxy-based Entity architecture in Meerkat ............................................................... 83
Figure 25: Extrapolation of entity position based on direction and speed .................................. 86
Figure 26: Dead-reckoning and local-lag implementation in Meerkat ......................................... 87
Figure 27: Snapshots of some Meerkat games ............................................................................ 94
List of Tables

Table 1: Total CPU time scores (milliseconds) ................................................................. 105
Table 2: Average memory usage scores (bytes) ................................................................. 108
Table 3: Average network data upload rate scores (kbps) ................................................... 110
Table 4: Average network download rate scores (kbps) ...................................................... 113
Table 5: Average frame-rate scores (frames per second) ..................................................... 115
Table 6: Average feedback time scores (milliseconds) ....................................................... 116
Table 7: Average feed-through time scores (milliseconds) ............................................... 117
Chapter 1

Introduction

Video games have become a major entertainment industry in recent years [1]. They span a wide range of platforms including PC, gaming console (e.g., Sony PlayStation, Nintendo GameCube, Microsoft Xbox), and handheld device (e.g., cell phones, PDAs, GameBoy) [2]. Game software has evolved and grown significantly to accommodate ever-growing feature and scale requirements of the multi-billion dollar industry. Production quality games often cost tens of millions of dollars to build, involving teams of more than 100 people [3].

The development of game software is one of the hardest aspects of building a game, due to its complexity and highly domain-specific requirements [4]. Major activities in game development include handling user input, applying game-logic or Artificial Intelligence (AI), physics processing, and game rendering.

Networked online games, a significant portion of the total game industry [5], involve an additional set of development challenges. Development activities of networked games include implementing the network communication architecture, distributing the game state and computation, replicating/synchronizing the game across clients, and ensuring game scalability. As concurrent distributed systems, networked games also have to deploy concurrency control and consistency maintenance techniques. Generic distributed system middlewares are not feasible options for games because they lack the domain-specific knowledge and optimizations required by games. Therefore, developers have to program distribution aspects specifically targeting at games.
Games can be thought of as real-time interactive simulations. Providing a smooth gaming experience to the player is a priority. For example, a First Person Shooter game may require an update rate of 30 frames per second for a correct and consistent game simulation. To ensure an acceptable level of fluidity and responsiveness, most games aim to render each frame within 16 milliseconds (providing 60 frames per second). Implementing game systems with such real-time performance requirements makes the development tasks described above even harder.

All such factors make game development a particularly complex activity. Developers require special domain knowledge and implementation skills to overcome these technical challenges. This barrier may prohibit non-expert game developers (e.g., hobbyists), or software developers from other domains, to build games for their needs (e.g., small/mid-scale games, games for research purposes, and educational games).

Even for non-beginners, taking care of low level development tasks requires significant effort and investment of time [4]. Simplicity and speed of development is particularly important for games. Fun and enjoyment, the main factors for a game’s success, are difficult to predict. The game development team has to rely on progressive introduction of new features and iterative refinement of them to test-out what makes the game more fun. As such, complexity of programming and the overall development time is one of the prime concerns in game development.

In recent years, the development of digital games has been simplified through the use of entity-based programming approaches. An entity is a game object that represents something in the game, e.g., avatar, vehicles, soldiers, trees, and monsters. In entity-based programming systems, the game is developed focusing around game entities, using abstracted entity-level constructs.
Entity-based programming tools include *Alice* [6], a tool for teaching programming to children, and the *Unreal Engine* [7], an engine used to produce professional quality games.

Usage of entity-level constructs implies that the developer does not have to deal with low-level technical details such as, physics processing, rendering, and raw networking. This simplifies the game system programming and quick addition of new game elements.

Another aspect of entity-based models is that the software is decomposed around *self-contained* game entities. Instead of having centralized routines for describing and managing the game, each entity contains its own description of state and computation including a description of its AI, response to user input, physics, interaction with other entities, and information for rendering. This facilitates the incremental and iterative development process games require, by localizing the code for each in-game entity and allowing the behaviour of individual entities to be modified in one place.

Up to now, however, entity-based tools have failed to appropriately address multiplayer games running over a network. At one extreme, we have tools (e.g., *Alice*) providing simplified and pure entity-based development, but with no support for network gaming at all. At the other extreme, we have tools (e.g., *Unreal Engine*) that do support distributed multiplayer games, but at the cost of breaking the entity-based model and requiring developers to handle low-level distribution concerns (e.g., description of data sharing, optimizations, and specification of remote procedures).
That leads to our central questions in this thesis:

- Is it possible to extend entity-based programming to networked games without breaking the model (i.e., revealing underlying distribution details)?
- Will such a system lead to games with adequate performance?

These questions led to our development of Meerkat, an extension of pure entity-based programming to networked games. The Meerkat toolkit provides a pure entity-based programming model that allows game development using entity-level abstractions. We have been able to extend the model transparently to distributed games where the developer does not need to write any additional code for networking. All complexities of game distribution are hidden from the developer.

This solution was achieved in three stages. First, we created a generalized concept of game entities. Entities in Meerkat are self-contained with descriptions of their state and behaviour (e.g., appearance, reaction to input, physics, logic/action, and AI). The “game” is simply a collection of all such entities where the gameplay emerges out of entity interactions.

After establishing the entity interface, we built generic algorithms for implementing game distribution. This generic system provides the foundation of distribution architecture for all Meerkat games.

Finally, we added an overlay of proxy objects on our architecture. This was done for two reasons:

- To hide the presence of distribution algorithms from the developer
- To tackle problematic scenarios that arise in fully automated networking
From the user’s point of view, a game programmer always deals with a proxy object for an entity without necessarily knowing how the entity will actually be distributed. The toolkit takes care of possibly complicated actions, for example, state access and interaction with entities in remote clients. The model ensures a seamless interface for developing both distributed and non-distributed games.

One major concern for such a system is its performance. Using custom-developed networking, the developer can implement a distribution system tailored for a particular game. It is possible to specify what parts of the state should be shared, and when state updates must be propagated, using in-depth knowledge of the game. Our automated networking system, however, does not have such privileges. It has to rely on the generalized entity interface and optimization techniques. An added factor is the use of proxy objects. Instead of direct object accesses, entity accesses are now intercepted and mediated by the system during the run-time.

To understand the performance overhead of our system, we created three games from three different genres. We developed two versions of each game: one used the automated networking providing by Meerkat, and the other one used game-specific custom distribution.

Our experiment results show that Meerkat games’ frame rate, feedback time and feed-through time were unchanged compared to hand-crafted networking versions. The amount of data transmitted over the network was in some cases as much as double using Meerkat. However, in other cases, the amount of data was dramatically reduced over hand-coding. This surprising result comes from opportunities to optimize network traffic that can be embodied in the toolkit when it has complete control over application’s networking. The overhead on CPU and memory usage was practically negligible. We concluded that for all but games that have the most exigent performance requirements, systems similar to Meerkat can be practical development solutions.
Our success in developing Meerkat and the findings of our tests show that it is possible to extend the pure entity-based approach to networking while still preserving sufficient performance.

The contributions of this thesis are:

- A pure entity-based model that works seamlessly in distributed games
- Implementation of a game development toolkit that shows the practicality of the model
- Identification and proposed solutions for overcoming some of the key challenges of transparent game networking
- Performance evaluation of fully automated distribution as compared to manual networking; impact analysis of different automation features (e.g., interest management and dead-reckoning)

The thesis is organized as follows. In Chapter 2, we provide a background of game development activities and entity-based development approaches. By discussing canonical examples of contemporary tools, we set the context of our work with Meerkat.

Chapter 3 presents the Meerkat model and explains its features. We present an example of developing a complete game with Meerkat, and show how it transparently extends to a networked multiplayer game.

In Chapter 4, we discuss how the model was implemented in Meerkat. We identify some of the key challenges of transparent networking and discuss how we overcame them.
Chapter 5 presents our evaluation of Meerkat discussing its practicality and soundness. We describe experiments for performance evaluation. First, we discuss the performance implications of major automation features in Meerkat. Then, we present an overall cost analysis of Meerkat, as compared to manual networking. Finally, we outline the limitations of our work and provide suggestions for possible future improvement.

In Chapter 6, we summarize the work and the findings of the thesis.
Chapter 2

Background

This thesis presents an extension of the entity-based programming approach to distributed multiplayer game development. To be able to describe our work and contributions more effectively, in this chapter we provide a general overview of multiplayer game development and how developers use existing entity-based tools to develop networked games.

Section 2.1 gives grounding on common game programming tasks and technical hurdles in developing multiplayer games. Section 2.2 discusses entity-based tools and solutions that simplify game development considerably by hiding various technical difficulties from the programmer. We give an overview of programming models of a few representative entity-based development tools, highlighting their strengths and weaknesses. We also discuss where Meerkat fits in the context of these works (Section 2.2.3).

2.1 Basics of Game Development

In this section, first we discuss the game development process and participants in a game development team (Section 2.1.1). Then, we take a closer look at general activities involved in game development (Section 2.1.2).

2.1.1 Game Development Process and Participants

Digital video games have become an important entertainment industry in recent years [1]. Game software has evolved and grown significantly to accommodate ever-growing feature and scale requirements of the multi-billion dollar industry. Production quality game projects now
require a collective effort of several heterogeneous teams. It is not unusual to have more than a hundred team members working for over three years to make a large-scale commercial game [8]. Among all the activities that go into making a game [9][10], we highlight three main categories below that are relevant to the discussions in this thesis:

**Game Design:** Game Design means defining and designing the actual game without giving much priority to the implementation details. Game design involves coming up with the game concept, game story, deciding on game actors (player and non-player characters/objects), gameplay, interactions, game objectives, and game mechanics.

**Content Creation:** Games require visual and auditory assets, such as, graphics, 3D models for actors, terrains, music, and sounds. Large teams have dedicated artists, level designers and animators for content creation.

**Game Development:** Game Development is the actual software development of the game. The goal is to implement the game that comes out of the Game Design process, using the contents/assets produced by the Content Creation team. Development activities generally involve designing the software infrastructure/components, processes, data models, and of course, doing the final programming. We discuss Game Development activities in more details in Section 2.1.2.

The activities of Game Design and Development tend to be interrelated and require high collaboration among these different teams [10]. Smaller teams might not have dedicated sub-teams for these activities. But, even then, these activities can still be discerned and they require isolated attention. Decisions made in Game Design can potentially have significant effect on how
the game will be developed. Conversely, several development constraints may also affect what can be done in the game, design-wise [11].

Fun, the main factor for a game’s success, is difficult to predict [12]. The team has to rely on progressive introduction of new features and iterative refinements on them to test-out what makes the game more fun [12]. As such, the agility and efficiency of the game development process is vital for the team’s success.

Now that we have some idea about where game development activities fit in context of the whole game creation process, we take a deeper look into some common game development tasks and related technical challenges.

### 2.1.2 Game Development Tasks

The features that need to be implemented in a game largely depend on the game at hand, the underlying system architecture, and performance requirements. The gameplay elements also vary across game genres [12][2][13][14] and designs, for example, depending on whether it is a racing game or a fighting game. However, there are common game software architectures and development tasks that transcend game genre. We present a brief discussion of them in two categories: General Tasks – that have to be done across all games (single or multiplayer), and Distribution Programming Tasks – activities and challenges particular to game distribution programming.

#### 2.1.2.1 General Tasks

A game can be thought of as real-time interactive simulation software. It is generally implemented using a central game-loop that possesses the main control of the program [15] [16]. This game-loop has the responsibility of generating (Update) and presenting (Render) the game.
to the player. Fast and repeated combination of these Updates and Renders gives the player a sense of continuous game progression.

The game-loop controls three major tasks:

- **Handling User Input**: This involves communicating with different input devices (e.g., keyboard, mouse, gamepad) and keeping track of their state changes.
- **Updating the Game State**: The game-loop advances the game state based on user input and game logic in each of its iteration.
- **Rendering**: This typically implies drawing the relevant part of game world to the screen using textures and models.

Chapter 4 has more details on the steps and process involved in a game-loop. Among these steps, rendering activities are often simplified by using graphics engines (e.g., OGRE [17]) or drawing APIs (e.g., Java 2D [18]). The majority of game development activities are related to the Update step, as we describe below.

Almost all the game logic is part of the Update section of a game. Game logic describes the behavior of all game elements. Input handling, i.e., how the game should react to user inputs (e.g., shoot a bullet if the player pressed the Space Bar key) is a major component of game logic. Game logic also describes game rules and actions (not necessarily player initiated) based on the time elapsed or other game events.

One significant portion of Update is doing the physics processing for the game world [11]. A racing game can be a good example which involves significant physics processing. Figure 1 shows a snapshot of the game Need for Speed. In this game, the player drives a car around a selected track. In its simplest form, the goal is to finish the track (or laps) before other players.
Several kinds of physics processing may need to be performed in a racing game as such. First, the moving car has to simulate its motion (velocity, acceleration, and deceleration). The car should not be able to go through road dividers. If it is raining, the road should be slippery and the car should skid. Finally, when colliding with another car, it has to show a bumping effect.

Another major computational activity in Update is Artificial Intelligence (AI) processing [19]. Any piece of logic to automatically control a game entity can be considered as game AI. An example of simple AI can be making a monster move around in the game world randomly or finding the shortest path to a destination. AI can also be complex, such as, planning a complete combat strategy for a computer controlled military base.
Figure 2: Screenshot of Age of Empires (an RTS game)

Figure 2 shows a screenshot of a Real Time Strategy (RTS) game, where an AI module has the responsibility of controlling player units, building structures, exploring the world, gathering resources and coordinating combat.

What makes game development activities like the ones mentioned above difficult to implement is the strict performance requirement of games. User input, game logic, physics, AI, rendering – all these have to happen fast so that the game simulation is fluid and responsive. For example, in a First Person Shooter game, it may not be unusual to require an update rate of 30 frames per second [11].
2.1.2.2 Distribution Programming Tasks

If we want to make the game playable by multiple players from different geographic locations, we have to develop a game distribution and networking system. A multiplayer game can be co-located (played in a single computer by people in the same room), but in this thesis, we refer multiplayer games as distributed multiplayer ones where players play on different machines over a network.

The majority of commercial multiplayer games use a client-server networking architecture for achieving centralized coordination, simpler implementation, better utilization of network bandwidth, cheat-prevention and easier concurrency/consistency maintenance [20][21][16]. The server can be fully-authoritative, meaning that the server has the sole responsibility of maintaining, updating and propagating game state changes to all the clients. The client is only responsible user input and game rendering according to the information it receives from the server. It is also possible to have a hybrid architecture where clients can take up partial responsibilities of game processing.

Depending on the scale of the game, we may have a cluster of servers instead of a single server – these are called multi-server games. Many large-scale games run on such array of servers which are highly scalable and fault-tolerant [22]. Our focus in this thesis is not on server side issues as such.

Previously, we defined game systems as real-time interactive simulation software. Now, we can add two more attributes for describing networked games – that they are concurrent and distributed. Developers face a set of additional complexities, for example:
**State and Computation Distribution:** Simply because there are multiple players controlling different entities, the game processing and rendering has to be distributed (at least between a central server and clients). Limited resources might be another reason to split up the game across multiple nodes so that the game does not depend on a single node’s processing or memory capacity [22].

**Synchronization:** As the state and computations are distributed, we have to keep all clients synchronized so that they can see and interact with the same view of the game world. If a client performs an action that modifies the game world, other clients must be synchronized to have an up-to-date state.

**Concurrency and Consistency Maintenance:** As all the clients interact with the same game world simultaneously, it naturally leads to concurrency issues where a shared object might not have any definitive state in a particular point of time [23][22]. This can potentially affect the integrity of some critical game entities, and thus, the whole game.

Even though the game state is changed continuously and concurrently, clients must be synchronized in near real-time [24][25]. As a result, games have to integrate consistency maintenance mechanisms to have an acceptable balance of fidelity (i.e., correctness of game state when player commits an action) and responsiveness [26][27][28].
Scalability: The game should scale well with an increasing number of players and/or increased game world size. If the game world is too big, there is no way a single client could keep up with the exponential growth of incoming information and the related processing [29]. Thus, multiplayer games have to deploy some form of interest management to filter out entities a client does not need to know about [30][22][26]. Even after that, there might be too much information to send back and forth if the nodes do not use advanced synchronization mechanisms like delta-encoding, where only the minimum chunk of information to denote the state change is transmitted [16]. Details about these techniques are in Chapter 4.

Starting from setting up basic network communication links, all these higher level distribution problems now also have to be handled by the developer. All such factors can make multiplayer game development a complex and time-consuming activity.

Using manual coding to handle distribution programming tasks (described above) can be a significant undertaking. Let’s take a simple example of object synchronization. We assume that a low-level network communication architecture is already in-place. Our only task is to listen for state updates (i.e., packets) coming from remote machines, and apply the changes to our local objects. For simplicity, we also assume that there is only one object in the game named Tank1. An example pseudo-code is given below to show how we may synchronize the tank object:
In this example, each attribute of the Tank object is updated one-by-one after extracting its value appropriately from the message packet. Manual programming as such allows fine-grained control over how the distribution is implemented. It also makes the best performance optimizations possible for a particular game. But, manual programming may become complex and error-prone as the game starts becoming more complicated. We can imagine how large and intertwined our code can become if we have fifty tank objects, hundred soldiers, and a large world-terrain where multiple players interact simultaneously. In addition to object synchronization, we also have to tackle other high-level distribution problems by hand, such as, state and computation distribution, concurrency control, consistency maintenance, and scalability.

One approach to simplify networked game development may be using distribution middlewares. General purpose distributed system supports include CORBA [31], DCOM [32], WCF/.Net Remoting [33], Java RMI [34], and Globe [35]. Although these systems provide various levels of support for implementing distributed shared objects, they tend to have a different focus than multiplayer game systems in terms of features and performance goals. Giving a “general” and transparent object distribution middleware is rather hard without having any
semantic knowledge about the objects. These systems fail to generally address and take advantage of system constraints that are specific to multiplayer games, for example, the fact that games do not require strict consistency for most game-objects [36]. A more detailed discussion on this topic is given in Chapter 4.

Networking and communication middlewares like RakNet [37] and HawkNL [38] specifically target games. But, the game developer still has to put a significant effort into implementing complex game distribution features without the help of game-level abstractions [15].

Most current game development tools recognize this need for providing game-level abstractions to simplify game programming. These tools provide entity-based game programming support that aim at hiding low-level implementation details from the game developer. In the next section, we define entity-based development, and then follow with a discussion of existing tools that support entity-based development. We present the strengths and weaknesses of different approaches, especially for developing distributed multiplayer games. Finally, we discuss where Meerkat fits in the context of current entity-based approaches.
2.2 Entity-based Game Development

An entity is a game object that represents something in the game, e.g., Soldier, Tank, Tree, Fort, Monsters, etc. Entity-based game development simply means that the game is written focusing around these entities, using abstracted entity-level constructs. Instead of writing, for example, a central routine for managing AI of all the game objects, entity-based development will spread out the AI calculations across all the entities. An entity thus becomes self-contained with a description of how to perform AI in that entity’s context. Similar to AI, the entities have concrete boundaries for all other game aspects as well: handling user input, description of physics, interaction behavior with other entities, and information on how it should be rendered on the screen. The “game” is simply a collection of all such entities where the gameplay emerges out of the entity interactions.

Of course, not everything in the game is conceptually an entity, for example, the world’s terrain and sky. For these cases, the development API may come with some special support for defining such non-entity elements, or it might be possible to implement these elements using entity structures. In either case, the core programming model can still be considered as an entity-based one.

From an implementation point of view, in a typical entity-based development system, each entity has some form of an Update function that is used to describe and control the action of that object in a particular frame. It can be thought of as the game logic fragment for that particular object. The main game-loop usually invokes the Update functions of every entity in the game so that each entity can update its state. When all entity Updates are done, the game has a new updated state as a whole. Similar to the Update steps, the game-loop may invoke individual Render functions for drawing out entities according to their specific requirements.
As entities are game domain concepts, game developers find it more natural to structure the game around these game-objects [29]. Clearly defined entity boundaries might help in localization of logic and state [29], allowing the behavior of individual entities to be modified in one place. As a result, the code may become more manageable with less code-coupling and inter-dependencies. This is particularly helpful for developing software like games, where code is often thrown away and the developers must take an agile and iterative approach.

As we now have a conceptual overview of entity-based systems, we move on to take a deeper look into some of the canonical examples of current entity-based development systems. First, we present some pure entity-based tools that successfully provide complete entity-level abstraction (Section 2.2.1). As we will see, current pure entity-based tools do not support networked games. Hence, we continue rest of our discussion by presenting entity-level tools that do support networked games (Section 2.2.2). As the topic of this thesis is game programming model, we focus on the programming models of these tools instead of their other features. Finally, we place our work in context of these existing entity-based approaches (Section 2.2.3).

2.2.1 Pure Entity-based Tools

We define the purity of entity-based approach by the degree to which it has been successful in presenting the game developer an entity-level abstraction. The developer should not need to know about low-level implementation details that do not belong to entity-level concepts e.g., how an entity actually achieves physics behavior using low-level concepts like bodies, shapes, and collision boundaries.
Kodu [39] is our first example of pure entity-based programming tool. Kodu takes a radical approach where the game itself is also the development environment. The visual programming interface that comes with the game implements a complete game programming language [40]. The language is aimed to be simple so that even new programmers like young children can develop games using an icon-based programming interface.

Kodu programming is entity-oriented where each character or object is programmed separately. Programming here means writing rules in the form of Condition and Action. For example, an avatar might have a rule like “when saw tree, move toward”. Here, “when saw tree” is the conditional and “move toward” is the action. Core constructs of Kodu programming are: Sensors (e.g., See and Hear), Filters (e.g., color of an object), and Actuators (e.g., Move). Figure 3 shows a screenshot of the visual programming interface, and code for an avatar’s behavior with...
three rules. Here, the first rule is game-logic/AI code that instructs the avatar to move toward a castle when it sees one. The second rule specifies a physics-response (jump) for the entity as it collides with a tree. This is an example of how Kodu abstracts physics processing details from the developer. The developer can simply write a behavior based on a high-level physics event like “bump”. The third rule specifies input-handling code where the avatar is being attached with arrow keys on keyboard (i.e., to control movement). This rule gives another example of abstraction where the developer does not have to deal with underlying details of input devices.

As we can see from the example, Kodu has very high level abstractions for providing fundamental game technology like collision detection, physics, and user input. It also has a wide array of prebuilt entities, sensors, and actuators to work with. But still, these constructs are limited as there is no scope for introducing user-defined custom entities. More importantly, Kodu does not come with any support for creating distributed multiplayer games.

Kodu is not the first of such pure entity-based approaches. In fact, Kodu is inspired [41] by the pioneer work laid out by Alice [6]. Similar to Kodu, Alice also comes with a visual programming interface. Programmers can create scripts by choosing visually presented phrases or code-blocks, instead of typing text code. Alice uses a visual scripting language for simplified object oriented design with simple behaviors and primitive object properties. Alice, however, primarily aims at teaching basic programming and 3D scripting to novices [6]. It may not be suitable for developing complex games [42]. As Kodu, Alice also does not support networked games.
Alice and Kodu have a strong focus on making the programming as easy as possible. Perhaps, that is the reason they use visual programming interface instead of a textual one. Whether visual programming is any more effective than textual, is a different topic altogether [43]. For the purpose of our thesis, we do not take the programming environment (textual or visual) into account, as long as the game is programmed around entities, using entity-level supports.

Other similar pure entity-based tools include Scratch [44], GameMaker [45], Cocos2D [46], Torque [47], GameSalad [48], and Artemis [49].

2.2.2 Entity-based Tools for Networked Games

Unlike the tools presented in the last section, this section presents entity-based tools that do have adequate multiplayer support. First, we discuss two game development toolkits (Unity and UnrealEngine) as examples of commercial entity-based tools. Then, we present two research projects (Journey and RTF) that successfully apply entity-based game distribution. We primarily focus on aspects of these tools’ programming model that are used for creating distributed shared game entities.

Unity

Unity [50] is a feature-rich commercial game development toolkit. It is a completely entity-oriented development platform where developers can use a graphical development environment for performing common development tasks. Every programming activity in Unity revolves around game objects (i.e., entities).
By attaching existing *components* to a game object, the developer can selectively reuse predefined functionalities that frequently occur in games. An example of a component is the Transform component which attaches relevant variables like Position, Rotation, and Scale to an entity. Similarly, components for Rigid Bodies, Colliders, Particles, and Audio are available. These components serve the purpose of providing entity-level constructs for game development. For example, after attaching Rigid Body and Collider components to a game object, Unity will automatically simulate physics for the object. Thus, the game developer is hidden from underlying physics implementation details.

In addition to using existing components, Unity game objects can also define new behavior by attaching scripts to the object. Those scripts may define how an object behaves in response to specific events. The developer may also define event handlers for that object, for example, *OnCollisionEnter* and *OnMouseOver*. The developer can create scripts for adding custom behaviors to game entities and utilize the expressive power of textual object-oriented programming.

We now take a look at how Unity supports game networking. For game distribution programming, Unity uses *State Synchronization* and *Remote Procedure Calls*. In order to make a game object use any of these, the developer has to attach a *NetworkView* component to that object where the developer specifies what kind of data should be observed for sending (i.e., which component), and how the data should be sent (e.g., Reliable Delta Compressed or Unreliable). If the observed component is a user defined script, the developer must explicitly serialize its data. Here is an example code-stub (from [51]) that shows Unity data serialization for an object’s script:
In the example, we see the game developer specifying synchronization for the `horizontalInput` variable. Here, the `horizontalInput` variable is used to capture a particular user-input named “Horizontal”. If the client is sending data, the game developer acquires the latest value using the Input.GetAxis( ) method and writes it to the stream. If the client is a recipient, the data is being deserialized into `horizontalInput` for possible future use.

Therefore, state synchronization in Unity requires explicit coding by the game developer. Even though the constructs are high-level, the game developer still has to think about how data should be serialized depending on the context (e.g., sending or receiving). The game developer also needs networking domain-knowledge to decide how data should be transmitted over the network, e.g., Reliable Delta Compressed or Unreliable. Thus, the simplicity and purity of entity-based model is compromised.

For using Remote Procedure Calls, the developer first has to tag the intended function with a “@RPC” attribute, and then has to pass information of that method to `networkView.RPC` function so that the method can be invoked. An example of how to use RPC in Unity is given below (from [51]):

```csharp
function OnSerializeNetworkView (stream : BitStream, 
    info : NetworkMessageInfo) 
{
    var horizontalInput : float = 0.0;
    if (stream.isWriting) { // Sending
        horizontalInput = Input.GetAxis ("Horizontal");
        stream.Serialize (horizontalInput);
    } else { // Receiving
        stream.Serialize (horizontalInput);
        // ... do something meaningful with the received variable
    }
}
```
In this example, the developer is writing an input-response code for the avatar. If the designated user-input for firing a bullet occurs (e.g., key press), the developer is firing a bullet using the `PlayerFire()` function. To replicate this method call on all clients, the `PlayerFire` function is declared as an RPC function, and it is invoked using a request through the `networkView.RPC()` call.

Again, we see that the game code is no longer in entity-level abstraction. The game developer requires knowledge about remote procedure calls, and the distribution of the game to decide which clients should replicate the function call.

**Unreal Engine**

The Unreal Engine is another popular game development tool. The Unreal Engine uses a similar entity-based model as Unity that is based around game objects. Therefore, we only discuss its network programming model. In UnrealScript (the scripting language for the Unreal Engine), the game developer has to create a separate code block for declaring the variables that should be replicated for an object. The developer also needs to specify under what conditions those variables should be replicated. We give an example code below (from [52]):

```plaintext
var playerBullet : GameObject;

function Update () {
    if (Input.GetButtonDown("Fire1")) {
        networkView.RPC("PlayerFire", RPCMode.All);
    }
}

@RPC
function PlayerFire () {
    Instantiate(playerBullet, playerBullet.transform.position,
              playerBullet.transform.rotation);
}
```

```plaintext
In this example, the developer is writing an input-response code for the avatar. If the designated user-input for firing a bullet occurs (e.g., key press), the developer is firing a bullet using the `PlayerFire()` function. To replicate this method call on all clients, the `PlayerFire` function is declared as an RPC function, and it is invoked using a request through the `networkView.RPC()` call.

Again, we see that the game code is no longer in entity-level abstraction. The game developer requires knowledge about remote procedure calls, and the distribution of the game to decide which clients should replicate the function call.

**Unreal Engine**

The Unreal Engine is another popular game development tool. The Unreal Engine uses a similar entity-based model as Unity that is based around game objects. Therefore, we only discuss its network programming model. In UnrealScript (the scripting language for the Unreal Engine), the game developer has to create a separate code block for declaring the variables that should be replicated for an object. The developer also needs to specify under what conditions those variables should be replicated. We give an example code below (from [52]):

```plaintext
var playerBullet : GameObject;

function Update () {
    if (Input.GetButtonDown("Fire1")) {
        networkView.RPC("PlayerFire", RPCMode.All);
    }
}

@RPC
function PlayerFire () {
    Instantiate(playerBullet, playerBullet.transform.position,
              playerBullet.transform.rotation);
}
```
In this example, we see that the game developer has to write conditions using constructs that denote whether the object has a new value (bNetDirty), who owns the object (bNetOwner), and the role of the client (Role). Access to these low-level distribution aspects imply a greater expressiveness and control over networking, potentially resulting in better performance for the game. However, the high-level nature of entity-based programming is getting violated by requiring the game developer to know about distributed object’s state (e.g., bNetDirty) and the distribution architecture of the game (e.g., bNetOwner and Role).

As the list of available game engines and tools is rather long (a sample list can be found at [53]), it was not possible for us to go through all of them and discuss their programming model. We believe these two commercially successful toolkits are representatives of similar entity-based tools that support networked game development.

These commercial tools are successful in presenting a pure entity-based model to the developer for single-player game development. However, to implement game networking and distribution, they start breaking the model by requiring developers to work with distributed...
system constructs. Moreover, if the developer wants to integrate advanced distribution features (details in Chapter 4) like extrapolation/dead-reckoning [54][26] and interest management, there is little entity-level support available in these tools. The game developers have to manually implement these high-level features and optimizations.

After the commercial game development tools, we now look at two research projects: Journey and RTF. These tools are not complete game development tools and have rather different focus. We include a discussion of them as examples of entity-based distribution systems.

**Journey**

Journey [55] is a multiplayer game development middleware that aims to hide complex issues of distribution programming from game developers. Journey provides abstractions for managing core networking tasks, such as, replication, interest management, fault-tolerance, load-balancing, and cheat-prevention. Duplicated game-objects are the core elements in this model. However, as in Unity, the game developer is required to provide annotations for specifying remote methods. Let’s take an example code (from [29]):

```java
@RemoteCall(dataSet="destination")
public void setDestination(CallContext callCon, double x, double y);
public Position getDestination();
```

In this example, the developer is specifying the `setDestination` method as a remote method (can be remotely invoked) by attaching a “@RemoteCall” attribute. We also see the use of other distribution level concepts, such as `dataSet` and `CallContext`. Datasets in Journey can be
considered as data-spaces. A remote method can only affect values of variables that share the same dataset as the method (details available in [29]).

In Journey, variables that need to be replicated are annotated using “@ReplicatedAttribute” tag. Replicated variables also need to be augmented with the information of which \textit{dataSet} they belong to. Below is an example (from [29]) showing variable replication in Journey using “@ReplicatedAttribute”:

```java
@ReplicatedAttribute(dataSet=“destination”)
protected Position destination;

@ReplicatedAttribute(dataSet=“speed”)
protected Position speed;

protected Path path;
```

Thus, Journey requires the game developers to work with distributed system constructs weakening the pure entity-based model.

RTF

RTF [15] is another tool that successfully utilizes data replication at the game-object level. When coding an entity, the programmer has to use custom data types provided by RTF. When the programmer extends entities from appropriate classes and uses these pre-defined data types, RTF can provide automatic entity serialization that is transparent to the programmer.

RTF, however, is primarily focused on simplifying multi-server development activities (e.g., zoning and instancing) rather than providing simplified abstractions for client development (i.e., game programming). RTF trades the purity of the entity-based model for the sake of greater flexibility and power. The game developer has to manually write the game-loop, interest
management policy, and invoke RTF functions at appropriate places so that the game state can be synchronized. As a result, the programming model is not a purely entity-based one in general.

2.2.3 Where Meerkat Fits

From our discussion of entity-based tools of varying purity, we see that pure entity-based tools (e.g., Kodu, Alice) have possibly the highest ease of use, where the developer does not require any special implementation knowledge other than the game concept (e.g., entity data structure and behavior). However, existing tools in this category have poor support for networked game development (if any).

We have some tools (e.g., Unity, Unreal) that are actually purely entity-based for single-player game development, but they start breaking the model as soon as we go into multiplayer development. The breaking of the model is significant in some tools (e.g., in Unreal), while it is relatively less in others (e.g., in Unity). We also see some works (e.g., Journey, and RTF) that simplify distribution aspects of multiplayer games, but they either have a different focus (e.g., multi-server development and scalability) or do not treat the purity of the programming model as their main concern. The simplicity and purity of the model is compromised for greater control over distribution.

While we agree that there are situations where flexibility and expressiveness are more important than the purity and simplicity of the model, we also believe that the developers should not be required to know about underlying implementation details in all cases. We are excited by the ease-of-use that pure entity-based models can provide, but we also think that pure entity-based models should not fail to handle distributed multiplayer scenarios. As we shall see, Meerkat fills this gap between pure entity-based models and models that provide multiplayer support.
Meerkat shows, in a proof of concept, that it is possible to provide a highly abstracted pure entity-based programming interface, while still keeping the network distribution completely transparent to the developer.

### 2.3 Conclusion

In this chapter, we have provided some basic idea about what technical challenges multiplayer game development (or game development in general) may entail. We then introduced entity-based game development approaches, which primarily aim at simplifying these difficulties for the game developer. After presenting examples of existing entity-based tools of varying purity and ease-of-use, we identified the importance of a pure entity-based system that seamlessly works in a distributed multiplayer setting as well. In the next section, we present such a programming model (Meerkat) and show how developers can use this model for creating different kinds of games.
Chapter 3

The Meerkat Programming Model

In the last section, we discussed the importance of pure entity-based programming models for simplifying game development. We also discussed how existing approaches either fall short on giving multiplayer support, or reveal distribution programming complexities to the developer. In this section, we present our solution, Meerkat, a pure entity-based model that also works seamlessly for creating distributed multiplayer games.

First, we give an introductory overview of Meerkat in Section 3.1. Then in Section 3.2, we describe the Meerkat model in detail by showing how we can use it to develop a distributed multiplayer game. Finally, in Section 3.3, we mention high-level controls present in Meerkat that can be used to adjust the distribution parameters of a game.

3.1 Overview of Meerkat

Meerkat’s purpose is to show that, after creating an underlying distribution system, it is possible to provide a game-programming interface to the developer that is purely at the entity-level of abstraction. Meerkat is not a complete (or polished) development environment. It also does not address game-system development activities, for example, load-balancing using multiple servers.

The central focus of Meerkat’s programming model is game entities. A game developer’s job is to identify the game’s entities and to create them using the entity-level abstractions provided by Meerkat. Each entity is fully contained with its description (e.g., AI, physics, and
input-response). Once the entities are programmed, the game at runtime consists of collection of entities, composed in a Scene. The gameplay emerges from the interactions of the entities.

To show the model’s practicality, we have implemented it under the .Net platform using the Microsoft XNA Game Development Framework [56]. We used C# as our programming language. General object-oriented programming constructs remain the same in Meerkat. This might give an impression that Meerkat (or entity-based models in general) is some form of object-oriented programming. But, we should recall from Chapter 2 that it is not necessarily so. For example, the programming model in Kodu is completely visual/icon-based. Unity uses another approach where entities are constructed from independent components. Thus, it is a particular case of our implementation that Meerkat resembles a typical object-oriented paradigm.

In the next section, we describe the Meerkat programming model by highlighting its core components for developing games.

### 3.2 Game Development with Meerkat

As it is difficult to discuss a programming model without referring to some concrete details, we will use our current Meerkat implementation to describe the model from a game-developer’s point of view. Throughout the section, we will use one sample game to discuss how Meerkat can be used to develop games from scratch. First, we describe the game’s design in Section 3.2.1. Then, from Section 3.2.2 through Section 3.2.7, we explain the core features of our model: entity classes, drawing and animation, entity state, physics, input-response, game-logic/AI, and entity interaction. Finally, we describe how to bring everything together to complete the game (Section 3.2.8) and make the game multiplayer (Section 3.2.9).
3.2.1 Snowballs – An Example Game

We will illustrate Meerkat through the example of a casual action game named “Snowballs”. In this game, each player controls an aircraft and tries to capture as many of a set of snowballs that fly around in the world as they can. To add more fun and action to the game, we will allow the players to shoot missiles at each other. If an aircraft is hit by a missile, the aircraft will become “numb” for some period. While numb, an aircraft may not grab any new snowball. To make our end goal clear, we give a snapshot of the completed game in Figure 4.
3.2.2 Entity Classes

Our first job is to decide which Entity classes our game will have. Entity classes are drawn from the game design. We find that we need entity classes for the aircraft, the snowball, and the missile.

Meerkat comes with a basic entity class (called Entity) that provides most of the common properties and methods to describe a Meerkat entity. We may simply start by extending from the basic Entity class. Figure 5 shows the entity-hierarchy for our Snowballs game. Snowball, Aircraft, and Missile classes extend the Entity class. We show the AutoMissile class as an example of multiple-level inheritance.

3.2.3 Entity Drawing and Animation

To describe how our entities should be displayed on the screen, we can use the Texture property inherited from Entity class. We simply need to set the Texture property with the name of the image file. Meerkat will take care of the rest of the drawing according to the entity’s up-to-date state (e.g., position, rotation and opacity). Here is an example where we specify the image file to be used for drawing the Aircraft entity:
In this example, we have an image file named “BasicAircraft.png” (a picture of an aircraft) inside the “Aircraft” directory. Meerkat will automatically load the image file and prepare necessary texture objects for rendering the entity.

Playing an animation for an entity is also simple using Meerkat. We need to provide an image file which contains several frames (i.e., snapshots). Meerkat draws the frames sequentially in a loop for rendering the animation. We have to set high-level animation parameters, such as, the number of frames present in our image and the desired speed of the animation, so that Meerkat knows how we want the animation to be played. To keep our example simple, we do not
use any animation in the Snowballs game. Figure 6 summarizes the members of the Entity class related to drawing and animation.

As we see, Meerkat is hiding all implementation details of drawing and animation from the developer, e.g., loading of image file, preparation of texture objects, management of the graphics device, positioning and rendering of texture, and animation implementation. Moreover, instead of maintaining any central data-structure or loop for rendering all the entities, game developers can describe each entity as a self-contained object using high level abstractions provided by Meerkat. To recall, this self-containment aspect and entity-level abstractions are vital for a pure entity-based model.

### 3.2.4 Entity State

The base Entity class provides most of the state variables needed by an entity as C# properties. C# properties are special accessor methods for private fields, but they can be used as regular data-members of a class. Properties provided by Meerkat Entity class may be used to access an entity’s position, angle/rotation, width, height, appearance (described in the previous section) and physics (described later in Section 3.2.5).

In addition to existing properties, custom fields and properties may be created for an entity to describe additional state variables. Variables that are used by an object exclusively (e.g., for internal calculations) should be created as fields, whereas variables that other entities may potentially interact with should be created as properties. For example, in our Snowballs game, we need a property for an Aircraft to denote whether it is currently “numb” (has recently been hit by a missile) and field variables to keep track of the time it must wait before shooting or getting damaged again (Figure 7).
3.2.5 Entity Physics

As discussed in Chapter 2, physics processing is one of the most difficult aspects of game programming. Physics processing involves collision detection, reacting to collisions and forces (e.g., showing bumping), and motion simulation for entities (e.g., updating the position of an entity in each frame according to its velocity). As we shall see, Meerkat automatically takes care of all such physics processing activities for the game developer.
In our game, we want an aircraft to collide with another aircraft so that they cannot pass through each other. To make an entity collide with others, we just need to select a collision-boundary shape for the entity using the CollisionShape enumeration. Possible values of the enumeration are: Rectangle, Ellipse, and None. Selecting Rectangle creates a rectangle shaped collision boundary around the texture of the entity. The entity will behave like a rectangular body during collisions. Similarly, selecting Ellipse creates an ellipse shaped collision boundary. The value “None” denotes that the entity should not exhibit any collision.

For additional control on the physics behavior, we may use properties to set bounciness, density, and friction for a particular entity. We may also subscribe to event notifications of when an entity collides or stops colliding with other entities.

Finally, Meerkat’s Entity class has properties that capture the entity’s physics simulation state. For example, the Velocity and AngularVelocity properties denote the current values of velocities of an entity (possibly affected by the physics simulation in the last frame). These
properties can also be set to new values to change how future simulation should be done. For example, setting a non-zero Velocity in an entity at rest will cause the entity to start moving. Figure 8 summarizes core members of the Entity class related to physics simulation.

As we see, Meerkat does not reveal any underlying detail of how physics is actually processed. The game developer deals with entity-level physics attributes only (e.g., Density, Friction, and Velocity). Meerkat automatically manages all necessary physics structures for the entities and keeps entity-states synchronized with the results of physics simulation.

3.2.6 Input Response, Game-logic, AI

As discussed in Chapter 2, the core gameplay of a game results from its game-logic, input-response, and Artificial Intelligence (AI) calculations. Input-response describes how the player controlled entities (e.g., avatars) acknowledge and react to user-inputs. On the other hand, non-player entities (entities that are not controlled by the player) have to have some form of AI or game-logic to describe its behavior in the game. Player controlled entities also have game-logic code to describe behavior/interactions that are not necessarily initiated by user-input.

In Meerkat, code for input-response and game-logic/AI is written by overriding the DoLogic() method of an entity. Typically, the DoLogic() method will be called at a rate of 60 times per second by the Meerkat engine’s game-loop. In each frame, the game “advances” according to the game-logic/AI rules described in DoLogic().

For our Snowballs game, the Aircraft entity has to have several user-input responses: moving or turning on arrow key-presses, and firing a missile when the space bar is pressed. Meerkat provides a high-level API for detecting user-input (e.g., whether a particular key has been pressed, released, or is being held down) that we can use here. The following code shows an
example of how we can make our Aircraft move (i.e., setting a new velocity) when the Up-Arrow key is pressed, and how we can shoot a missile when the space bar is pressed:

```csharp
// Aircraft.cs
public override void DoLogic()
{
    if (Scene.IsKeyPressed(Keys.Up))
    {
        _vx = (float)Math.Cos(Angle);
        _vy = (float)Math.Sin(Angle);
        Velocity = new Vector2(_vx, _vy) * _speed;
    }
    if (Scene.IsKeyPressed(Keys.Space))
        Shoot();

    // ... other game-logic
}
```

Here, `Shoot()` is an example of a custom-method that we wrote for the Aircraft. We may create such custom-methods to better organize our code-base.

As AI code is game-logic code for computer controlled entities, we use the following AI example for showing how we may write game-logic/AI for an entity. In our Snowballs game, the Snowball entities are computer-controlled and they should fly around randomly on their own. We may achieve this behavior by continuously moving the snowballs around the world, and by changing their direction randomly after an interval. For interval calculations, we can use the `ElapsedTime` property from Scene which denotes the time elapsed since the last frame (i.e., since the last invocation of `DoLogic()`). The code is given below:
Here, we keep a timer variable (_rotateTime) to check whether enough time has passed (i.e., two seconds). Once two seconds have elapsed, we shift the direction of the Snowball by a random angle and update its Velocity to reflect the change.

It is not necessary for all game-logic to go in the DoLogic() method. In fact, a major portion of game behavior is usually initiated by physics events, such as, collisions of two entities. For our game, we want that if an aircraft touches a snowball (i.e., physics collision), the snowball should update its current owner to the aircraft and its color should change to that of the aircraft. We override the OnCollisionWith() method of Snowball to describe this behavior. When the Snowball collides with any entity (i.e., Meerkat invokes OnCollisionWith() method of Snowball), we check whether the entity is an Aircraft. If it is an Aircraft, we change the owner (GrabbedBy) and color (Hue) accordingly:

```csharp
// Snowball.cs
public override void Dologic()
{
    _rotateTime += Scene.ElapsedTime;
    if (_rotateTime > 2f)
    {
        _rotateTime = 0;
        Angle += MathHelper.PiOver2 * random.Next(-1, 1) / 10f;
        _vx = (float)Math.Cos((double)Angle);
        _vy = (float)Math.Sin((double)Angle);
        Velocity = new Vector2(_vx, _vy) * _speed;
    }
}
```
3.2.7 Entity Interaction and Communication

We have already seen how entity collisions (e.g., between Snowball and Aircraft) may act as a trigger for entity interactions (e.g., Snowball accessing the Aircraft’s hue). In Meerkat, cross-entity communication/interaction is done through properties and methods of the target entity, but not field variables. An entity may access (i.e., get, set, invoke) the properties and methods of another entity provided there is adequate privilege (e.g., member visibility).

For our Snowballs game, when the player presses the space bar key, we want the Aircraft to shoot a missile. The Aircraft entity has to interact/communicate with a newly created Missile entity to properly set the missile’s position and heading (to match those of the Aircraft), and then to launch the missile by invoking its Start() method, as shown below:

```csharp
// Snowball.cs
public override bool OnCollisionWith(Entity other)
{
    if (other is Aircraft)
    {
        grabbedby = other;
        Hue = other.Hue; // change color
    }
    return false; // do not show bumping
}
```

All entity interactions start with first having a reference to the target entity. We have seen how an entity’s reference can be discovered through method arguments (e.g., in the
OnCollisionWith( ) method), or as the return value resulting from the creation of an entity (e.g., as used in the Shoot( ) method to initialize the variable m). In addition, an entity may also do queries to the Scene to get references of in-game entities.

One such query is GetEntitiesInRage( ) which can take a center (i.e., position) and radius as parameters and return all entities (i.e., references) within that zone. Another way is to go through the Entities list in the Scene, which is a directory of all entities present in the game. For example, if an aircraft gets a special weapon that can automatically shoot nearby opponents within ten meters, we can use the Entities list to find targets to shoot at. We can go through all the entities in the Entities list and check which of the aircrafts are within ten meters, as shown below:

```csharp
// Aircraft.cs -> DoLogic()
foreach (Entity e in Scene.Entities)
{
    if (e is Aircraft && e != this
         && Vector2.Distance(e.Position, Position) < 10f)
    {
        ShootTowards(e);
    }
}
```
3.2.8 Bringing it Together

We have now seen all the major Meerkat constructs that we can use to build our entities. After all the entity classes are created, we compose a Scene class for the game that extends from the base *Scene* class. This custom scene class is the game-class that brings together all entities into the game (Figure 9). The Meerkat toolkit provides a coding framework where we only need to specify which class is the Scene. Meerkat creates an instance from that scene and starts the game execution.

In the custom scene class, the developer distinguishes between activities that exclusively belong to a particular player and activities that belong to the global description of the game. To describe global activities that occur when the game loads, we use the *InitializeGame()* method. This method will include, for example, creation of non-player entities (e.g., snowballs) and their preparation. In our Snowballs game, we use this method to create fifty snowballs and position them side by side:

```csharp
// AircraftGame.cs
protected override void InitializeGame()
{
    for (int i = 0; i < 50; i++)
    {
        SnowBall s = CreateEntity<SnowBall>();
        s.Position = new Vector2(i, 0);
    }
}
```

Similarly, player-specific entity creations and initializations are done in the *InitializePlayer()* method, for example, creating an avatar. The entities created inside this method are considered to be belonging to a particular player. For example, in a real time strategy game, soldier units and the base-camp can be considered belonging to a player, whereas, trees and animals belong to the global description of the game (part of *InitializeGame()*).
Inside the \texttt{InitializePlayer()} method, we may also use the scene-level game services provide by Meerkat to initialize other player settings. For example, if we wanted to attach a camera to our \textit{Aircraft} so that the client-display always has the avatar \textit{Aircraft} at its center, or if we wanted to simulate a sky background for the game, we could use the \texttt{Camera} and \texttt{CreateBackground()} services. The following example shows the \texttt{InitializePlayer()} method for the Snowballs game, where we create an \textit{Aircraft} avatar, prepare the game camera to follow our aircraft, and create a sky background:

```java
protected override void InitializePlayer()
{
    Aircraft avatar = CreateEntity\textless\texttt{Aircraft}\textgreater();

    // Setting up the camera
    Camera.Target = avatar;
    Camera.TargetFOV = 10; // zoom-level

    // Using a texture pattern of clouds to create the sky
    // Creating a 10x10 grid for the pattern, starting from Position(-10, 10)
    CreateBackground("\texttt{Aircraft/cloudsFew}", 10, new \texttt{Vector2}(-10, 10));
}
```
Figure 10: Core members of Entity and Scene classes
Finally, the `PlayerSceneLogic()` method can be used to describe scene-level logic that does not necessarily belong to any entity, for example, input-handling for exiting the game. Similarly, GUI drawing code (e.g., for showing the player number on display screen) can be written inside the `DrawGUI()` method.

To summarize the Meerkat API, we present the core members of the `Entity` and `Scene` classes in Figure 10. Throughout this section, we have discussed how these Meerkat classes provide entity-level properties and methods for developing a game. Starting from managing the basic game-loop, Meerkat takes care of all implementation details of entity/object management, drawing and animation, physics processing, and input-handling. This entity-orientation and abstract entity-level constructs make Meerkat a pure entity-based programming model.
3.2.9 Making the Game Multiplayer

In this section, we discuss the core focus of this thesis – how to make distributed multiplayer games. So far, we have seen how to construct entities and how to bring them together in a Scene. To make the game multiplayer, the developers actually do not need to do anything in addition.

There is no concept of “developing the server” for a Meerkat game. The code we have developed so far for the Snowballs game is enough to make the game multiplayer. Meerkat provides generic implementations of the game server and clients (details in Chapter 4). As discussed in the previous section, the game developer only has to create game entities and compose the scene.

All entities will automatically be replicated and synchronized on all the clients. Unlike other entity-based tools (Chapter 2), game developers do not need to specify which entities/properties/fields should be shared, and how (e.g., network transmission method). New entities may be created (e.g., the missile) or existing entities may be destroyed (e.g., an Aircraft is blown up). Players may join or leave any time. However, currently there is no fault-tolerance support (e.g., handling node disconnections) which can be implemented in future.

Meerkat provides basic concurrency control and consistency maintenance for the game. It does not provide strict concurrency or consistency features (e.g., atomic transactions) for reasons described in Chapter 4. Meerkat will distinguish between player entities (e.g., Aircraft) and non-player entities (e.g., Snowball) for proper network distribution. For player entities, Meerkat will make sure that each player has exclusive control over their units (e.g., a player may not control other players’ avatars). Meerkat achieves this by assigning entity ownerships to clients for carrying out computations (details in Chapter 4).
Entities can interact and communicate with other entities that are possibly on different client machines, for example, a missile from an aircraft (i.e., belonging to a player) may hit and cause damage to another aircraft (i.e., belonging to another player). Physics simulations will also continue to work for cross-client entities (e.g., car collisions in a racing game). Meerkat continues to support property accesses and method invocations on remote entities without requiring any additional code from the game developer. However, the current implementation of Meerkat restricts property access to read-requests only. Property writes (i.e., state mutation) on remote entities are not supported for consistency reasons (details in Chapter 4). This may imply some inconvenience of coding since property values of remote entities cannot be altered directly. However, the same effect (i.e., state mutation) can still be achieved through a method call on the remote object where the method performs the desired state mutation (e.g., assigning a new value to the property).

As we have discussed in Chapter 2, multiplayer games have to deploy various optimization techniques (e.g., interest management, delta-encoding, and dead-reckoning) to ensure scalability and game responsiveness. Meerkat provides built-in implementations for several of such major optimizations. To reduce bandwidth usage, Meerkat implements interest management, extrapolation/dead-reckoning, selective-sharing, and delta-encoding. To increase game responsiveness and consistency, Meerkat also provides the local-lag [57] feature. Chapter 4 presents a detailed discussion on how Meerkat integrates various optimizations inside the toolkit.

In Meerkat, entities may keep references to other entities even if they are on remote machines. For example, if an aircraft stores a reference to another aircraft for launching an attack in the future (e.g., when conditions are more favorable), the aircraft can continue to use the reference (e.g., for periodically checking the opponent’s position) even when the opponent
aircraft is far away in the game-world. This transparent entity-referencing is one of the major strengths of Meerkat compared to other entity-based tools, especially in the presence of network optimizations (e.g., interest management). More details on this topic are in Chapter 4.

To emphasize the fact that Meerkat maintains the same pure entity-based model (Section 3.2.8) even when there is full distribution, the complete code for the *Missile* entity is given below as an example:

```csharp
[Serializable]
public class Missile : Entity
{
    float _speed, _lifetime, _vx, _vy;
    public virtual int HitPoint { get; set; }

    public Missile()
    {
        Texture = "Aircraft/missile";
        CollisionShape = CollisionShape.Rectangle;
        HitPoint = 1;
        _speed = 5f;
        _lifetime = 2f;
    }

    public virtual void Start()
    {
        _vx = (float)Math.Cos(Angle);
        _vy = (float)Math.Sin(Angle);
        Velocity = new Vector2(_vx, _vy) * _speed;
    }

    public override void DoLogic()
    {
        _lifetime -= Scene.ElapsedTime;
        if (_lifetime <= 0) Destroy();
    }

    public override bool OnCollisionWith(Entity other)
    {
        if (other is Aircraft)
        {
            (other as Aircraft).TakeHitFrom(this);
            Destroy();
            return false; // no bumping effect
        }
    }
}
```
The example shows a complete entity with its texture and physics description, game-logic code to self-destruct, collision and cross-entity communication for damaging a possibly remote aircraft, and a custom *HitPoint* property that the affected aircraft can use to calculate the damage. As we see, we did not have to put any additional code for achieving game networking. The distribution programming is completely shielded from the game developer.

### 3.3 High-level Parameters for Controlling Distribution

In the previous section, we saw how we can create a complete multiplayer game without being exposed to any networking details at all. We also mentioned some major optimizations that are present in the toolkit.

However, sometimes the game developer might want to have more control over how distribution takes place. For example, the developer might want to prevent some custom-defined fields/properties of an entity from being shared to further reduce bandwidth consumption. Another example can be a client’s interest-radius that dictates how close other remote entities have to come before the client starts receiving updates for those approaching entities (details in Chapter 4). A default interest-radius that works well for a Real Time Strategy (RTS) game might not be a good choice for a racing game. In the RTS game, the avatar movements are comparatively slower, allowing more time to load new game entities. On the other hand, a racing car moves comparatively faster, and the game world needs to be prepared well in advance.

We recognize the need for having such controls that can potentially improve game performance, depending on a game’s type and design. At the same time, we also do not want to walk away from the ease-of-use that purely abstracted entity-based systems can offer. In Meerkat, we offer some high-level parameters that can be adjusted to improve performance without dealing
with low-level distribution programming concerns. The parameters are examples of how such controls may be presented to the developer. Below we present three such parameters.
**Figure 11**: Two games using two different interest radius values

**Interest Radius**

The radius of a client’s interest-zone may be controlled using the *InterestRadius* property available in *Scene* (Figure 10). In Figure 11, we see two games using two different interest radius values. The one on the left (radius = 5 meters) will have a smaller area of interest compared to the one on the right (radius = 10 meters). Smaller area of interest may imply lower bandwidth consumption. For example, the client on the left only receives updates of entity E1, whereas, the client on the right receives updates of E1, E2, and E3. However, the client on the left might suffer from game responsiveness if it suddenly moves towards E2 or E3 (E2 or E3 might not get loaded fast enough).

**Local Lag**

Meerkat uses local-lag [57] for improving consistency maintenance across clients. Local-lag is a technique where the execution of an entity’s state change (or rendering) is artificially delayed so that the change information can first propagate to other clients before it is committed.
It helps all the copies of the entity remain more consistent across clients. In Meerkat, the developer can easily use the *Lag* property of Entity (Figure 10) to set the amount of local-lag desired.

**What Gets Shared**

To selectively exempt some fields/properties of an entity from being shared with remote clients, we can put the *[Player]* attribute on the fields/properties we want to exclude. The code below shows an example where we use the *[Player]* attribute on the *_lifeTime* field and the *HitPoint* property to exclude them from being shared with other players (i.e., clients). The *Speed* property will be shared on all clients.

```csharp
// Missile.cs
public class Missile : Entity
{
    [Player]
    float _lifeTime;
    [Player]
    public virtual int HitPoint { get; set; }
    public virtual float Speed { get; set; }

    // ... rest of the class
}
```

As we see, the distribution control parameters in Meerkat are mostly simple *properties* (except for the *[Player]* attribute). These parameters do not require any complex network programming knowledge. Even then, we should re-emphasize that these parameters are only *optional* and the game developer is not *required* to use these controls.
3.4 Conclusion

In this chapter, we have described the pure entity-based model of Meerkat and its current implementation from the game developer’s point of view. With example code, we have illustrated Meerkat’s various components and how to create a complete multiplayer game without writing any networking code at all. In the next chapter, we will describe how we implemented the Meerkat toolkit, reviewing its core design, and presenting key challenges that we had to overcome to provide fully automated networking.
Chapter 4

Implementation

In this chapter, we describe how we implemented the Meerkat game development model presented in Chapter 3. In the first part of this chapter (Section 4.1), we describe the basic Meerkat engine that realizes the Meerkat programming model and supports single-player games. In the second half (Section 4.2), we describe how we extended the single-player implementation to support distributed multiplayer games maintaining the same programming interface for game developers. Along the way, we discuss key challenges of transparent network distribution and how we overcame them. Finally, we discuss why we implemented our own distributed entity system instead of using generic distributed object middleware, in light of the discussions presented in this chapter.

4.1 Meerkat Model and Single-Player Engine

The implementation of Meerkat was done under the .Net platform using the Microsoft XNA Game Development Framework [56] and C#. Later in this chapter, we will progressively introduce a few other tools that we used as development supports (e.g., physics engine and networking middleware). It should be noted that the Meerkat model is not inherently dependent on these tools. We could have possibly used a different graphics engine, game development library, physics engine, and networking middleware to provide another implementation of Meerkat. The development supports used for our current implementation are simply our preferences.
In the following subsections, we describe how we started with the basic facilities provided by the XNA framework and gradually added our components to implement the Meerkat model.

4.1.1 The Game-loop and Key Elements

The XNA framework provides facilities for implementing a game-loop. XNA provides an Initialize( ) method that game programmers override to initialize game settings, classes and game assets (e.g., loading textures). The Update( ) and Draw( ) methods make up the core of an XNA game-loop. Game programmers override the Update( ) method to provide game logic for entity movement, physics, player input, and AI. Similarly, the Draw( ) method is overridden to specify how game entities and backgrounds are displayed on the screen.
Figure 12 shows the basic Meerkat game-loop realized through the XNA game-loop. XNA invokes the Initialize( ) method only once when the game starts. After that, XNA goes into an infinite loop of Update( ) and Draw( ) calls so that the game state can be continuously updated and rendered. An update step followed by a draw step is called a frame. As updates or draws might take different times to complete in different frames, the time between subsequent Update( ) calls may vary. This is referred to as variable time stepped game-loop.

Theoretically, we could just have used this basic XNA game (involving one class) for our entire game. But, we want a clear separation among the tasks done by a Client, services of a game Scene, and description of the Entities (as described in Chapter 3). Accordingly, we break down the class structure in Meerkat into three major classes: Client, Scene, and Entity.

The Client class is the XNA game class that we started with (i.e., the class that describes the game-loop). Since we wanted to integrate our Scene and Entity objects into the game-loop, we propagated the Client’s Initialize( ), Update( ), and Draw( ) calls across Scene and other Entity
objects. For that purpose, we appointed Scene as an intermediary between the game Client and Entities for easier management of game objects.

Figure 13 shows the modified game-loop of Meerkat that breaks down the execution flow and management across three roles: Client, Scene, and Entity. The Client forwards the Initialize( ) and Update( ) calls to Scene so that Scene can perform game-specific initializations and updates. After its update, Scene invokes the DoLogic( ) methods of all in-game entities so that the entities can execute their share of game-logic/AI (as discussed in Chapter 3). Scene takes care of drawing the entities, GUI, and backgrounds which liberates the entities from game rendering activities (details on the next section).

4.1.2 Drawing and Animation

XNA provides a high-level drawing API for rendering textures into the display. We need to instruct XNA as to which texture we want to draw (i.e., graphics file) and how we want it to be drawn (e.g., the position on screen, texture rotation, scaling, and tint). Drawing an entity requires Meerkat to call the Draw method of XNA with appropriate values taken from the attributes of each entity. Meerkat loops through all the entities to draw them one by one.

If an entity is playing an animation, Meerkat must determine the correct texture source (i.e., frame) among the multiple frames contained in the entity’s texture file. This calculation is done by using the animation attributes of that entity, e.g., animation speed, time elapsed, and animation mode. After all of the entities are drawn, Scene draws the GUI for the game, which is described by the game developer in the DrawGUI( ) method. The game rendering steps in Meerkat are summarized in Figure 14.
4.1.3 Adding Game Physics

XNA does not provide built-in support for physics processing. The burden is on the game developer to write all physics processing code inside the Update() method. We wanted to provide physics out-of-the-box in Meerkat to provide entity-level physics support for Meerkat games. For this purpose, we use the Farseer physics engine [58] to carry out physics simulations for our entities (i.e., collision detection, physics response and motion simulation).

Even after using a physics-engine, there is a significant amount of work involved to perform physics simulations. The game-world and the physics engine’s internal world (“physics-world”) are two separate spaces. For each of the game entities, we need to create physics structures (e.g., bodies, fixtures, and joints) inside the physics-world and initialize them with physics attribute values (e.g., mass, friction, and velocity). Then, in each frame, we have to instruct the physics engine to advance its simulation for the time elapsed since the last frame. In
each frame, we have to make sure that the entities in the game-world and their corresponding physics objects in the physics-world are kept synchronized.

In Farseer, we have to create and maintain two objects of type *Body* and *Fixture* to represent one physical body in Farseer’s physics simulation world. Meerkat’s programming interface does not reveal the presence of such underlying physics objects to the game developer (as was described in Chapter 3). Meerkat maintains the physics structures in background and automatically synchronizes the simulation results with game entities without requiring the game developer to write any code to do so.

The properties that are required by Farseer Body and Fixture objects are mostly one-to-one mapping with how we defined our Entity class interface (showed in Figure 15). However, some features required more work for Meerkat, for example, the OnCollision event. The OnCollision event raised by Farseer notifies about two Farseer Fixtures that have collided. We had to translate this information from Fixtures to Entities so that the method reports Entities that have collided, instead of Fixtures. Thus, the game developer can work with entity-level objects instead of underlying physics structures.

When an entity is created (discussed later below), we create the necessary Body and Fixture objects for the entity taking values from its physics-related attributes. We also have to create a collision boundary shape taking the CollisionShape, Height and Width values of the entity. If the developer did not specify any value for these attributes, we assign some default values so that the game can start with basic physics functionalities.

After we have created and initialized physics structures for an entity, we have to integrate the physics processing into our main game-loop so that the physics simulation advances each frame of the game. We assign this job to the Scene class as it acts as the central coordinator.
The results of physics simulation may introduce changes to entity states. For example, if the physics engine is simulating motion of a moving entity, the entity’s Position, Velocity, Angle, and AngularVelocity may be constantly updated by the physics engine. Therefore, before letting the entities execute DoLogic() methods, the Scene makes sure that physics simulations has been properly done in current frame and the entities have an up-to-date state to work with (Figure 16).

After all entities have executed their DoLogic() methods, their states may have changed. For example, an entity might have been assigned an increased velocity as part of its game-logic. The physics engine is not aware of such changes because these changes are done on Meerkat Entity objects. Therefore, the Scene has to add an additional step to synchronize Entity objects with physics engine structures. After this synchronization, the physics engine has the up-to-date state of each entity and the simulation can continue in the next frame (Figure 16).
4.1.4 Input Handling

XNA provides a high-level API for detecting user input from keyboard, mouse and gamepad. For example, we can use the IsKeyDown( ) method to query whether a particular key on the keyboard is being held down in the current frame. In games, generally the developers are also interested in knowing whether a key has been pressed just in this frame (to mark the start of an action) or whether a key has just been released (to indicate that an action has stopped). To offer such higher level methods (e.g., IsKeyPressed, IsKeyReleased), Meerkat maintains two snapshots of the input device states: one from the previous frame and one for the current frame. By comparing these two states, Meerkat is able to tell the exact change that happened in the current frame (e.g., key pressed, key released, mouse button pressed, mouse button released).

The Meerkat Scene class coordinates these input handling activities and provides high-level methods (e.g., IsKeyPressed( ) ) to the game-developer (Figure 17).
4.1.5 Entity Creation, Spawning, Deletion

So far, we have described how Meerkat implements:

- Game-loop execution
- Overridable default implementations for Scene initialization and entity DoLogic( )
- Automatic drawing and animation of entities
- Automatic physics processing for entities
- High-level mechanisms for input handling

To make our model and single-player engine complete, we just need to add entity creation, spawning, and deletion features.
In the previous chapter, we discussed the `CreateEntity()` method that the game developer uses to create entities. The spawning of an entity (i.e., its inclusion in the game) is actually automatic because Scene starts to process the entity (e.g., rendering and physics) as soon as it is created (described below).

Figure 18 outlines the steps involved in Meerkat entity creation. First, Meerkat takes the user specified class type and creates an object of that type. In order to make all game entities globally identifiable, Meerkat assigns a unique ID to the entity. After loading the texture file of the entity, Meerkat prepares its physics structures (already discussed in Section 4.1.3). Next, the entity is registered with Scene so that it can be managed by Scene from now on. Finally, a reference to the object is returned so that the entity state can be accessed by the developer.
Deletion of an entity is similar but reverses the steps outlined above. Entity deletion involves unloading the texture file, destroying the physics structures, and unregistering the entity from Scene.

We have now completed the Meerkat programming model and single-player game engine. Throughout this section, we have discussed how Meerkat implemented the game-loop, drawing and animation, game physics, input-handling, and entity life-time management for providing a pure entity-based model to the game developer. We move on to the next section for describing how we extended our work to add distribution support for multiplayer games.

4.2 Meerkat Multiplayer

In this section, we give an overview of Meerkat multiplayer implementation. First, we describe the basic client-server networking architecture in Meerkat that is used for exchanging messages/commands between the server and the clients (Section 4.2.1). Once the basic command passing architecture is established, we describe the implementation of the entity-replication system in Meerkat (Section 4.2.2). In subsequent sections, we describe the remaining distribution features (Section 4.2.3, 4.2.4) showing how they maintain a pure entity-based model for the developer (Section 4.2.5). Finally, we discuss some of the optimization techniques that we implemented (e.g., dead-reckoning, local-lag, and delta-encoding) for improving multiplayer game performance (Section 4.2.6).
4.2.1 Client-Server Architecture for Exchanging Commands

In Meerkat, we wanted to use a client-server networking model so that we can conveniently control the overall game distribution from one node. To recall from Chapter 2, most games use client-server models for achieving centralized coordination, simpler implementation, better utilization of network bandwidth, cheat-prevention, and easier concurrency/consistency maintenance [21][16]. However, we do not use a fully-authoritative server that is responsible for processing the entire game. As we will see later in this chapter, Meerkat uses a hybrid model of state and computation distribution across the clients and the server. The Meerkat server’s purpose is to provide a single point of contact for all the clients, and to implement a centralized game coordination.

All communication between the server and game clients takes place through some fixed format message/commands. We use a Command class that describes, in one object, the message/command and any additional parameters it might require (Figure 19). The CommandName field denotes the command type (e.g., CreateEntity, RemoveEntity, ClientLeft) and the Args array may be used to pack an arbitrary number of arguments (i.e., argument objects), depending on the needs of a particular command.
The basic networking infrastructure consists of a central server, game clients, and the communication channel of commands/messages between a client and the server. In the first half of this chapter (Section 4.1), we have already discussed how we created the clients for single-player games. Now, we have to modify the Client’s game-loop so that it can listen for incoming commands from the server. The client should also be able to send out commands to the server as needed. The server program is a loop that listens for incoming commands from the client, processes them, and sends out any required commands/responses to the clients (Figure 20).

For establishing the communication architecture in Meerkat, we used the Janus Toolkit [59] as our networking middleware. Even though Janus is mainly used to implement our entity replication and consistency maintenance system (will be discussed later), we can also use it to implement the client-server architecture we need for communicating our Command objects.

All communication in Janus is done through the Timeline objects that Janus provides. Timelines can be treated as message “channels” to which nodes can subscribe for publishing messages and receiving updates. To establish a complete bidirectional channel between a client and the server, we create two Timelines for each client at the Meerkat server: one for sending out commands and one for receiving (Figure 20).

As the network loop in Janus runs in a different thread than the main application (i.e., Meerkat game-loop), it can potentially interrupt our normal game processing. Commands from the server may arrive at any time and a client might not want to execute the command right away if it is in the middle of some other critical operation. For example, the server may give a command to destroy an entity that the client is currently doing physics calculations with. To avoid such conflicts, we maintain a queue to buffer incoming commands and pull them only at a discrete point in our game-loop when it is safe for the game. Our Incoming Command Handler
Figure 20: Meerkat client-server communication architecture using Timelines
delegate helps this process by queuing up incoming commands and providing thread-safe access to the commands queue (Figure 20).

Figure 20 shows the overall client-server communication architecture we implemented using Timelines. Any client can now send and receive Command objects to/from the Server. A discussion of this architecture was critical because throughout the rest of the section, we will refer to various kinds of commands that clients and servers exchange. When we refer to such a command, it should be implied that we already have the mechanisms of creating, transferring, and responding to such commands.

4.2.2 Entity Replication

Earlier in this chapter (Section 4.1), we discussed how self-contained entities can be added to the Scene, automatically creating the gameplay. For a multiplayer game, we now have to replicate and synchronize these entities across all the clients so that clients can interact with these entities as part of a shared virtual game-world. For example, in our Snowballs game, all the players see the same game view (e.g., same set of snowballs flying around at the same locations) and can interact with them (e.g., grab) as part of one game-world.

For entity replication, we again use the Janus Timelines that we introduced as part of the client-server command architecture (Section 4.2.1). The Janus Timeline Server that provides support for realizing Timeline structures is hosted as a component of the Meerkat Server.

To recall, Timelines can work as channels for communicating objects over the network. When a new value (i.e., object) is written to a Timeline, the new value is broadcast to all nodes that have subscribed to that Timeline. Any subscribed node can invoke a Get operation on the Timeline at any time, to read the latest value that was broadcasted in this channel.
Now, if we reserve one Timeline per object, we have essentially created a way to share that object across all the nodes. Any node can get the latest value of that object from the Timeline, mutate the object locally to produce a new value, and then write it back to the Timeline so that other nodes also get this new value.

We use exactly this method to replicate our game entities. Whenever a new entity is created on a client using the Scene.CreateEntity() method, the client also creates a new Timeline to share the entity with other clients. In our client-server model, a Meerkat client does not know about other clients that are present in the game. So, to share the entity with other clients, the client that creates an entity informs the server about this event. This is done by sending an AddEntity command to the server that includes information about what entity was created and the ID of the Timeline that will broadcast updates on this entity. The server relays this information to other clients in the game so that they can create the entity locally and subscribe to its updates.

Similarly, when a client deletes an entity, it broadcasts a RemoveEntity command (through the server) so that other clients can unsubscribe from that entity’s timeline and destroy associated local objects (e.g., physics and texture).
Figure 21: Entity replication architecture in Meerkat (shown for two clients)
As the clients maintain a local copy for each entity in the game world (Figure 21), the clients must also synchronize these local copies in each frame before interacting with them. The synchronization of local copies is done following the steps below in a client’s game-loop (also shown in Figure 21):

a. At the very beginning of a frame, Scene grabs the latest values from all entity Timelines and updates old copies of entities with these new values.
b. Scene now proceeds with its normal operations (e.g., physics processing and game-logic) on these fresh local copies.
c. When the all updates are done, Scene writes these local entities into their timelines to broadcast possible state changes that happened in this frame.

For drawing entities, Scene always uses the local entity copies which are already up-to-date after steps (a) to (c) above. The entity replication architecture described in this section is summarized in Figure 21.

4.2.3 Entity Ownership and Computation Partition

In the architecture we discussed in the previous section (Figure 21), all the clients perform simultaneous computations on all game entities (e.g., physics simulation, input handling and game-logic). There are three major problems with this approach:

First, replicated computations might introduce ambiguity in game-logic. Let’s take an example of avatar entities (e.g., Aircraft entities are avatars in our Snowballs game). Since all the entities are replicated to all the clients, any client actually has full access to all avatars in the game. Now, if a client feeds the user input to all avatar entities it has access to, it will appear as if this client is controlling all the avatars itself. Similarly, other clients will also try to take over
control of all avatars when the input-response section in DoLogic() gets executed on those clients. Obviously, in our game we do not want a player to be able to take over control of other players’ avatars. Thus, we have ambiguous and erroneous gameplay taking place.

Second, simultaneous updates to an entity by different clients may cause significant inconsistencies. There is always some lag in the network and if multiple clients are allowed to modify the state of the same entities (e.g., by running game-logic and physics simulation), we have a much increased chance of introducing noticeable inconsistencies.

Third, we actually do not need to burden all the clients with performing the same computations that will eventually produce the same results. For example, all the clients do not need to simulate the motion of a moving entity. Only one client needs to do the CPU intensive calculations and let others know the results (e.g., position updates).

To sum up the three points above, replicated computation done for an entity in multiple clients is redundant at best, detrimental at worst. If a game still requires computation replication for some other reasons, e.g., to minimize network traffic or to improve user-response time, there are various techniques [60][61][62] that can be adopted for more robust concurrency control and consistency maintenance. However, as concurrency and consistency maintenance policies come with a price on performance, most games are not strict about them for general entity replication [27][29]. For these reasons, we implemented a straightforward policy of entity-ownership in Meerkat for providing basic concurrency control and consistency maintenance.

In our entity-ownership based system, the client that first creates an entity is declared the owner of that entity. The owner is the only client that can change the state of the entity. Other clients may still read the entity state anytime, but they may not modify the entity’s state. Computations for an entity (e.g., DoLogic() and physics) are also exclusive rights of its owner.
In Figure 22, we show the example of entity ownership for our Snowballs game. There are two players controlling two aircraft, and two snowballs are flying around. Client1 and Client2 own the aircrafts A1 and A2 respectively. The Missile that is created by A1 is also owned by Client1. The snowballs S1 and S2 got created as part of the `InitializeGame()` method of the Scene. To recall, this method is used to describe global elements and settings of the game that does not exclusively belong to any player. For handling such global entities, the Meerkat server
runs a dummy client named *Client0*. *Client0* is like any other regular Meerkat client, except it does not need to perform any input-handling or game rendering. Its only responsibility is to maintain the global entities that are controlled by AI.

We also have to make some changes to how physics processing is done when we want to have entity ownership policy in place. Previously, all the clients would simulate physics for all the entities. Now, we need to skip physics processing for entities that a client does not own.

We could achieve it simply by not creating any physics-related structures for un-owned entities. This approach works fine if we are only talking about motion simulations. But, we will have problems if we want physics reactions (e.g., bumping) between two entities that are owned by two different clients. For example, in Figure 22, the Missile M1 is owned by *Client1* and the Aircraft A2 is owned by *Client2*. If each client maintains physics information for their owned entities only, there is no way to detect and raise the collision event between missile M1 and aircraft A2.

We overcame this problem by creating lightweight physics objects that are bare minimum to just detect a collision for non-owned entities. Beyond this, we instruct the physics engine not to perform any other simulation (e.g., motion and bumping) that are redundant or may lead to modification of entity states. Thus, even though *Client2* does not maintain full-featured physics structures for missile M1, it maintains the minimum information so that a collision can be recognized.

After such inter-client collisions takes place, we also make sure that a client does not invoke `OnCollisionWith( )` and `OnSeparationFrom( )` methods on entities that it does not own. This is because the client that actually owns those entities will call these methods. We do not want potentially ambiguous behavior by invoking these methods multiple times.
4.2.4 Interest Management

We introduced interest management in Chapter 2 (Section 1.1.2.2) as a common optimization technique that multiplayer games deploy [22]. Even though it is an optimization technique, we discuss it here because of its significant influence on Meerkat implementation (discussed in the next section).

Interest management is a technique to reduce bandwidth where instead of sending updates for the whole game world, a client is sent information about only the entities that it is “interested” in. A widely used criterion for determining interest is proximity [27], where we treat entities in close proximity (i.e., determined by positions) as potentially interesting for a client. There are various interest management algorithms of varying complexity and efficiency [30]. We implement a Euclidean Distance Algorithm [30] based interest policy in Meerkat.

In our policy, we take the positions of entities that a client owns as center points of interest zones. For each of these points, we check which entities are close enough (determined by the interest radius) and consider them within interest. In Figure 23, we give an example of interest calculation for a client that owns entities A1 and M1. Taking these two entity’s positions as center points, the server determines that this client should have A1, S1, S2, S3, M1 and A2 in its interest set. All other entities (e.g., S4, S5, and S6) are considered out-of-interest. We expose the interest radius property as a distribution parameter that can be adjusted by the game developer (as discussed in Chapter 3).

Recalling from Section 4.2.1, the Meerkat server is a simple loop for processing and forwarding commands to the clients without the need to know anything about the game. But for interest management, the server now has to have knowledge about all game entities. Though it
is not necessary for the server to know the complete game state, it must know *enough* about the entities (e.g., ID, position and ownership) so that the interest sets can be calculated.

Since we are already running a dummy client (Client0) at the server side for managing global entities (discussed in Section 4.2.3), we utilize this client to make our interest management implementation simple. We integrated Client0 with the server so that the server could access the game state through this client. We also created exception rules in interest management for Client0 to make sure that all game entities remain inside its interest set. Finally, we made the interest calculation a part of the main server-loop to calculate client interests in each frame (we could also have done it less frequently to save CPU processing).

The server always maintains a copy of what every client already has in its interest set. At each frame, the server computes a new interest set for each client and compares it with their
previous state from the last frame. With this comparison, the server can identify the entities a client should create (because the entities came close enough) and the entities that a client should destroy (because they went too far). The server sends out AddEntity and RemoveEntity commands to the clients for carrying out these changes. The client already knows how to handle these commands since they are the same commands that get used during remote entity creation and deletion (discussed in Section 4.2.2).

4.2.5 Keeping the Networking Abstract

The multiplayer system we have developed so far using generic entities has two major weak points that threaten its functionality, as described below:

Problem 1: Entity references in automated networking

Recalling from Chapter 3, Meerkat allows game developers to use entity references irrespective of whether the entity is local or remote. Let’s take an example game code:

```csharp
// Aircraft.cs
public override bool OnCollisionWith(Entity other)
{
    if (other is Aircraft)
    {
        myEnemy = other;
    }
    return false;
}
```

In the example code above, when an Aircraft entity collides with another Aircraft, the game developer stores the entity reference as `myEnemy`. This reference might be needed sometime later in the game, for example, to initiate a revenge attack on `myEnemy` when the time is opportune. It may happen that the target Aircraft for which the developer stored a reference goes too far away from this client after some time (e.g., the enemy aircraft went to a different part
of the world). The interest management system (Section 4.2.4) will then issue a command to this client to destroy the seemingly out-of-interest entity. If the client uses this reference to read the state of the entity (e.g., position), it will receive stale values as the client has already unsubscribed from the entity’s updates. If we implemented Meerkat with C/C++, we would also have manually de-allocated the entity object and the game could possibly crash when the client accessed this reference. We need to give a solution for tackling this problem to provide a robust and transparent interest management system.

There is another case where entity references become problematic. In our entity replication system based on Timelines (discussed in Section 4.2.2), Scene has to update all the entity objects each frame with new values from the Timelines. Meerkat does this by discarding the old object and acquiring a new one from the Timeline. Therefore, the object reference stored by the game developer becomes obsolete as soon as Meerkat changes the object inside. While this particular case may be applicable to our use of Timelines only, generally, we need to find a solution to ensure that the underlying engine can operate on entities without affecting the user references.

**Problem 2: Method invocations on distributed entities**

As discussed in Chapter 3, the Meerkat user is shielded from the fact that an entity is distributed. That means that we may have situations where the user code in one client invokes methods on entities that actually do not reside on the same machine. This will still work for methods that only rely on reading entity states, since all remote entities are updated in each frame. But this will not work for methods that also need to mutate the remote entity because the client does not have write privileges for objects it does not own (discussed in Section 4.2.3). We need a way to
intercept such method calls and relay this invocation request appropriately to the client that actually owns the entity.

In summary, we needed a system for safely referencing distributed entities and intercepting accesses on them so that we can continue providing abstracted networking for the user. We achieved it by using a proxy-based architecture (Figure 24). Whenever the developer creates a new entity (using Scene.CreateEntity( ) method), instead of returning a reference to the actual entity object, we create an additional proxy object for the entity and return a reference to that. This proxy object acts as a mediator between the developer and Meerkat engine. It can intercept accesses made by the developer and forward them to the internal Entity or to a remote entity as appropriate (more on this below).

We used the Castle Dynamic Proxy library [63] for creating our proxy objects and to specify the interception logic. Using Castle, we can create proxy entities during runtime while still retaining the same interface (and implementation) as user provided custom entities. Proxy objects created using Castle can intercept method accesses only. Property accesses (i.e., Get/Set) are like regular method calls under the hood, and as such, we can intercept property reads and writes of an entity. But, there is no way to intercept field accesses as they do not go through any intermediate method calls. This is the reason Meerkat does not support fields for cross-entity interactions (described in Chapter 3). We show our proxy-based entity architecture in Figure 24.
Using the proxy-based model, the Meerkat engine can now perform necessary networking operations on the internal entity object while keeping the proxy object intact. This solves the entity referencing problem we were having earlier for updating entity states from Timelines. The Scene can now replace the internal Entity object with a new value from the Timeline without affecting the proxy object that the user has references to.

Unfortunately, solving the problem with automated interest management is not that straightforward and we require some additional changes. If an entity goes beyond the interest zone of a client, the server now sends a *HibernateEntity* command instead of the previous *RemoveEntity* command. Upon receiving a *HibernateEntity* command, we can destroy the internal Entity safely since the Meerkat engine itself does not need the entity anymore (e.g., for physics processing and drawing). However, we still have to protect the proxy object because the user might have saved references to this and it may be accessed sometime later.
At first, it may seem like a significant overhead on system resources since we cannot completely destroy the proxy objects for entities that go out-of-interest. But, the proxy entities are lightweight, as they do not need to maintain any engine-related structures (e.g., physics structures, and texture). Additionally, to save computational cycles, we specially mark these objects as *hibernating* to skip them from regular computational steps in the game-loop, e.g., physics simulation and rendering.

Only in the case when the user code accesses a *hibernating* entity, we prepare the entity on-demand by loading its up-to-date state from the Timeline. We intercept the access (e.g., Get of a property) and redirect it to the newly acquired copy of the entity so that the user gets a fresh value. We intercept Set accesses done on properties of remote entities and prohibit them from proceeding. This is done irrespective of whether the target entity is *hibernating* or not, according to our consistency maintenance policy described in Section 4.2.3.

Finally, we add interception logic to properly support method invocations on remote entities. After interception, we check whether the client owns the target entity, and if so, we allow the invocation to proceed. If the client is not the owner of the entity, we send a *MethodInvocation* command to the server containing information about the ID of the target entity, the method name, and method arguments. The server takes responsibility of properly forwarding the request to the client that actually owns the entity.

Creating such method invocation requests, executing the method on a remote machine, and finally getting back the results over the network can be time consuming. If we were to allow remote methods with return values, we would have to block the client that makes a remote invocation request until the results come back. For such reasons, most games do not expect a
return value on remote method execution [29]. We wanted to prioritize game performance (i.e., frame-rate) over the feature-richness of remote methods, and as such, we do not allow remote methods to expect return values in current implementation of Meerkat. We made all remote method invocations non-blocking (i.e., asynchronous) so that the client can continue its normal operations without affecting the frame rate.

### 4.2.6 Distribution Optimizations

In previous section, we described our basic implementation of Meerkat multiplayer that can be used for developing networked games. We have also already discussed how we integrated the optimization technique of interest management in Meerkat. Even though we may have functional games using the implementation done so far, we need to add several other widely used distribution optimization techniques (e.g., Dead-reckoning [54], Local-lag [57] and Delta-encoding [16]) for general scalability of Meerkat games. We present such optimizations in the following subsections:

#### 4.2.6.1 Dead Reckoning and Local Lag

Dead-reckoning is a technique where instead of communicating all entity state changes, the clients deploy some form of prediction to interpolate/extrapolate the state of an entity. Let’s take an example scenario involving two clients (Client1 and Client2) and an entity (E1). Suppose Client1 is simulating the motion for E1 by updating E1’s position in each frame. Even though the state of E1 is getting changed each frame in Client1, we do not need to send out all the changes to Client2. Client2 can actually predict future positions of E1 by extrapolating E1’s last received position and its recent velocity (speed and direction).
In Figure 25, we show two position updates of E1 that a client received at time T1 and T2. We can use these past positions to calculate E1’s speed and the direction it is heading. Based on that, we can predict E1’s position at a future time T3.

The clients need to have a synchronized clock and run the same prediction algorithm in both sides for a proper dead-reckoning system. When Client1 has a new value for E1’s state, it runs the prediction algorithm locally to see whether Client2 can successfully predict the same state. If the states match within an acceptable tolerance level, Client1 can skip sending the update because it knows that Client2 can successfully generate the new state locally.

Dead-reckoning is mainly applied for entity fields that get updated frequently but still maintain a smooth progression, e.g., position and angle [64]. The technique will obviously fail if the entity changes its speed or direction abruptly (e.g., during sharp turns). In that case, we must send the new state of the entity. But for most entities that move in a smooth motion, the dead-reckoning technique can significantly reduce the network bandwidth usage [29] as we only need to send a fraction of all updates.
In addition to reducing bandwidth usage, dead-reckoning may also improve the quality of simulation. Instead of waiting for all state updates to arrive, a client can continue drawing remote entities by doing interpolations and extrapolations on the known state values as appropriate.

To provide additional support for improving consistency in the presence of network lag, we also use Local-lag. In Local-lag, whenever a client has a new state for an entity, the client waits briefly (e.g., 25 to 100 milliseconds) before actually committing the change to its local entity. In the mean-time, the client broadcasts the change information to other clients so that all clients can apply the change together, at approximately the same time.

In Figure 26, we outline the Dead-reckoning and Local-lag system we implemented in Meerkat. To recall, we write new state values of all entities that a client owns after the updates finish in the client’s game-loop (described in Section 4.2.2). At that stage, we invoke the
Timeline Set operation with a new entity value and the amount of local-lag we want. Our networking middleware Janus invokes Meerkat-provided Extrapolate() and Difference() functions to check whether it should proceed with the writing. Our Extrapolate() function extrapolates the position and angle of the previously sent entity. The Difference() function visits all fields of these two entities and checks whether there is any significant difference in any field. If there is a significant difference, Janus proceeds with the writing operation by automatically taking care of the local-lag that needs to be applied.

4.2.6.2 Compacting Serialization and Integrating Delta Encoding

Meerkat uses the default serialization method for XNA and C# data structures. For Meerkat Entities, it provides a custom serializer and deserializer for efficient data transfer over Janus Timelines. It should be noted that Meerkat takes care of implementing this custom serialization process. The game developer does not have to worry about serialization.

In custom entity serialization, we first specify which members are client-specific and therefore should be exempted from serialization. Most of the fields of an entity class fall into this category, for example, loaded texture assets and physics structures. Since physics calculations for an entity are done only by the client that owns the entity, we also skip physics attributes (e.g., Density, Friction, and Bounciness) from being synchronized with other clients.

There are some attributes for which deciding what to synchronize can be tricky, for example, entity animations. If we try to synchronize an animation frame by frame across clients, the network lag will lead to choppy animations on the receiver side. For this reason, we only synchronize the attributes that succinctly describe the animation an entity is playing (e.g., texture file name, total number of frames, and animation speed) and leave out other details (e.g., exactly what frame the animation is currently on).
The final category of fields that we can exempt from serialization is the fields that contain a `[Player]` attribute tag (a high-level distribution parameter in Meerkat, described in Chapter 3). We can identify such fields using C# reflection and skip them.

In addition to filtering out unnecessary attributes that do not need to be synchronized (discussed above), Entity references are another reason for which we needed custom serialization. Since entity references in Meerkat are actually references to proxy objects, we needed to give special care for its serialization process. If we encounter a field that is an entity reference type (i.e., reference to a proxy object), we replace the field value with the ID number of the entity that the referred proxy object actually represents. On the receiving side (during deserialization), we check for fields that contain entity IDs and replace them with the actual entity objects (i.e., proxy entities on that client). As entities are replicated in all the clients, this ID based replacement works nicely for implementing a global entity referencing system.

We applied the final level of network data optimization by implementing a delta-encoding [16] system for communicating entity state changes. Without delta-encoding, a client would need to write the complete state of an entity to the Timeline even if only one field had been changed. In our delta-based system, when an entity needs to be written to the Timeline, we first go through all of its fields and check which fields actually changed since the last broadcast. We pack only the information of those fields that changed and write that in the Timeline. Other clients receive this delta-encoded update and apply the patch to their existing entity copies for reconstructing the complete new state of the entity.
In addition to the optimizations we already discussed in this chapter, we also integrated several other engine optimizations to improve our engine/game performance. They include minimizing the number of interceptions on proxy objects, caching the read/write results from Timelines and the use of method delegates instead of reflection whenever possible. As these optimizations are not core focus of the thesis, we omit their discussion here.

4.3 Conclusion

In this chapter, we have discussed how we implemented the Meerkat model and extended it to support distributed multiplayer games. We identified some key challenges (and presented our solutions) of transparent distribution implementation in a pure entity-based environment.

After our discussion in this chapter, it might be easier to understand why we implemented our own distributed entity system instead of using a generic distributed object middleware (e.g., CORBA, DCOM, WCF/.Net Remoting, Java RMI, and Globe). If we look back, there were various instances where we utilized our knowledge of data and usage patterns in games.

The first example is our utilization of discrete steps in a game-loop. We saw that the only time game developers need access to the game state is during the DoLogic() execution. Therefore, we could freeze the object state (from the engine’s perspective) and do all necessary housekeeping (e.g., physics processing and network synchronization) before or after DoLogic().

Interest management was possible only because entities had a spatial notion. To a client, some entities were “close” and others were “far”. A generic distributed object system would not be able to consider objects with such “positions”.

Dead-reckoning, one of our major optimizations, was possible because objects followed a predictable path. Overall, we took advantage of loose consistency requirements of games [36] in
several other aspects of our implementation. All such optimizations would not be feasible with a
general purpose system that lacks the semantic knowledge about games. However, it can be a
future work to compare performance of a generic distributed object middleware with a domain-
specific system.

In the next chapter, we evaluate Meerkat’s success as a pure entity-based system and
estimate the overhead of using fully automated networking provided by Meerkat.
Chapter 5

Evaluation

In Chapter 2, we introduced entity-based game development and illustrated that almost all current game development tools follow an entity-based approach. We also presented canonical examples of existing tools, to analyze their success in abstracting networking through entity-level constructs. From our discussion, we identified the lack of an entity-based game development model that also seamlessly extends to distributed games.

In light of this, we investigated two key questions in this thesis:

a. Is it even possible to provide and implement an entity-based game development model that also works transparently in a distributed setting?

b. Will an implementation of such a model be practically usable without unreasonable costs on performance?

In this chapter, we present our evaluation of Meerkat that addressed the above two questions. First, we will discuss how successful Meerkat has been in terms of practicality and soundness of the model (Section 5.1). Then, we will take a look at the performance of our Meerkat implementation, to understand possible performance implications of such models (Section 5.2). Finally, we will present a discussion of Meerkat’s limitations and possible future work (Section 5.3).
5.1 Practicality and Soundness of the Model

Our first check is whether Meerkat is a pure entity-based model to start with, even for single-player games. This is verified in detail in the first part of Chapter 3 (Section 3.2), where we presented the model and explained how Meerkat allows entity-level description of game-logic, physics, input-handling, and overall game rendering.

The second check is whether Meerkat successfully extends the entity-based model to distributed multiplayer games. This is illustrated by showing that Meerkat does not require any change or additional code for making the game multiplayer (Chapter 3, Section 3.2.9).

To show that the Meerkat model can be practically implemented in a game engine, we developed a complete implementation of Meerkat and discussed how such a system may provide automated and transparent networking for distributed games (Chapter 4).

In Chapter 3, we have already presented a casual action game that we developed using Meerkat (the Snowballs game). Later in this chapter (Section 5.2.1.1), we will describe two other Meerkat games from other game genres: a Real Time Strategy (RTS) game (Figure 27.a) and a racing game (Figure 27.e). Previously, in a different project [65], other developers used Meerkat to create multiplayer games from several other genres, such as, a tower-defense game called Monkey Apocalypse (Figure 27.b), a platformer game called Spikeball Stadium (Figure 27.c), and a sandbox game called Liberi Building (Figure 27.d). Figure 27 shows snapshots of these games (a snapshot for the Snowballs game was already given in Chapter 3). We take Meerkat's ability to develop all these different games as an indicator of its general expressiveness and power.
a. The RTS game

b. Monkey Apocalypse (tower-defense)

c. Spikeball Stadium (platform, side-scroller)

d. Liberi Building (sandbox)

e. The Racing game

Figure 27: Snapshots of some Meerkat games
Finally, we would like to mention another different project, Liberi Live [66], where a level designer and game editor was built around Meerkat. Liberi Live indicates that it is possible to create game-authoring tools for Meerkat to push it towards making a complete game development system.

5.2 Performance of the Implementation

To understand how Meerkat did in terms of performance, we created three sample games for performance testing. We were generally interested in finding the overhead of Meerkat’s automated networking as compared to manually coded game distribution. The purpose was to see whether a system like Meerkat will be practically usable in terms of performance. We also performed tests to understand performance implications of different automation features in Meerkat (e.g., interest management and dead-reckoning). The description of the tests (Section 5.2.1) and our findings (Section 5.2.2 and Section 5.2.3) are described below.

5.2.1 Method and Setup

In our discussion below, we will present the conditions of our experiments (Section 5.2.1.1), the metrics that we assessed (Section 5.2.1.2), and our procedure for data collection (Section 5.2.1.2).

5.2.1.1 Conditions

There were three dimensions over which we varied our experiment conditions: the game genre, level of automation in networking, and network latency.
Game Genres

Different genres generally involve different styles of gameplay [2][14], and consequently, different pattern of load on system resources. We created three games from three popular genres [2][13]: casual action, Real Time Strategy (RTS), and racing – that covers such variances on system load (described below).

The casual action game is already discussed in Chapter 3 as the Snowballs game. To recall, in this game, each player controls an aircraft to grab snowballs that are flying around in the world. An aircraft may also shoot missiles on opponent aircrafts. For our test’s purpose, we had fifty snowballs in total spread out in the world. Thus, this game gives us moderate paced entity movements and collisions, a medium-size game-world, and a large number of dynamic entities producing significant network traffic.

In the RTS game, each player owns a fort and ten soldiers. The goal for a player is to discover where other players are and destroy their forts. The world terrain is made up of trees (500 of them). Thus, this game is populated with a much larger number of entities.

The player controls the soldiers using the mouse: left mouse button to select (or group select) soldier(s), right mouse button to specify the target location. The soldiers automatically move to the specified target by calculating the shortest path and avoiding obstacles, leading to CPU intensive AI calculations.

Upon sight of an opposing unit (i.e., soldier or fort), the player can issue an attack command to its soldiers (that are already selected) by right clicking on the opponent unit. The attacks are automatic from thereon. The soldiers move to the opponent unit and start the attack (i.e., fighting animations will be played). The attacking soldiers cause “injury” to their target, meaning that the attacked unit will keep losing its health gradually. If the health of a unit reaches
zero, it is considered dead (e.g., soldier become dead, forts become destroyed). The player to survive till the end of the game (e.g., all other players’ forts have been destroyed) is considered the winner. There are ten relics spread out randomly in the world. If a player grabs a relic, all its soldiers are given a health and hit-point boost.

The relatively slow-placed gameplay involves determining a strategy for exploration (e.g., finding relics), defense (protecting own fort) and attack (destroying opponent forts/soldiers) given the limited resource (i.e., ten soldiers). Overall, this game is likely to reveal a new set of performance characteristics (than the Snowballs game) as the game-mechanics, computations, and entity-types are considerably different here.

The third game is a car racing game. The player controls a car using the keyboard (i.e., accelerate, brake, and steer left/right) to move through a given track. The track has obstacles like trees, walls and road-blocks. The cars show collision and bumping effects when they hit obstacles (or other cars). The player to finish four laps of the track first is declared the winner. The racing game serves as a different kind of game involving fast input-processing, CPU intensive physics calculations, and high frame-rate requirements (for rendering both local and remote entities).

**Level of Automation**

All of our performance tests aimed to find whether the performance overhead of Meerkat’s automated networking is reasonable. We can test this by developing the game networking using Meerkat, and then by not using Meerkat to compare performance differences. We consider the code developed using regular Meerkat as fully automated since the developer does not need to do any manual network coding. We denote this condition by Auto. The other condition is called Manual, where the network synchronization part is manually developed without using Meerkat’s automations.
To keep our comparisons fair, we were careful not to change too many factors between the *Auto* and *Manual* conditions. For this reason, we defined the *Manual* condition as follows:

The *Manual* coding condition uses the same game engine and client-server command passing architecture (described in Chapter 4) as Meerkat. However, we disable the automation features that come built-in with Meerkat: entity replication/synchronization, interest management, and dead-reckoning/extrapolation. It means that we will no longer use Timelines to replicate entities, and the entities will be regular objects instead of proxies. This will allow us to find the overhead of using proxy objects, interceptions, and Timeline-based state replication. It will also help us evaluate the effectiveness of our generic distribution algorithms (e.g., state and method replication, interest management, and dead-reckoning) that previously determined what is shared and when, without any knowledge of the game.

Disabling automatic entity replication means that game developers now have to manually broadcast new values when the entity states get changed. This is achieved by rewriting the Set methods of entity properties that we want to share with other clients. In addition to setting a new value to the local entity’s field, we now send a command to the server containing information about the target entity, field-name, and the new value. The server broadcasts this command to all other clients so that they can update their local copies of entities that changed.

Let’s take an example of the Position property. Previously, the Position property was straightforward, relying on automation algorithms, as shown below:

```csharp
public virtual Vector2 Position
{
    get { return _position; }
    set { _position = value; }
}
```
In the Manual version, we rewrote the Set operation to add code for sending a SetPosition command to the server containing the entity-ID and the new value. We also check whether the change of value is significant enough to broadcast. The code is shown below:

```csharp
public Vector2 Position
{
    get { return _position; }
    set
    {
        if (Vector2.Distance(_position, _lastPosition)>0.001f)
        {
            SendServerUpdate(new Command() {
                Command = "SetPosition",
                Args = new object[] { ID, value.X, value.Y }
            });
            _lastPosition = value;
        }
        _position = value;
    }
}
```

Similarly, we also rewrote the entity methods that we want to replicate on remote clients. We included additional code to create and broadcast appropriate messages to denote game events as they occurred. For example, if an Aircraft entity grabs a Snowball (determined inside the OnCollisionWith method), we now create a SnowballGrabbed message containing information about which aircraft grabbed which snowball (e.g., SnowballGrabbed Aircraft2 Snowball39). Thus, the distribution is now game aware, sending data for only the states/events we are interested in, and only when they are relevant.

In addition to finding the general overhead of Meerkat (Auto) compared to manually coded networking (Manual), we were also interested to see the effects of some of our major automation features: interest management, dead-reckoning, and the [Player] attribute (that can be used to specify custom sharing). For each of these, we wanted to see whether they had a significant effect on performance (or how much), to justify their inclusion in Meerkat. That gave
us three more conditions where we turn-off each feature one by one. For example, from the fully Auto condition, we disable the interest management feature and get a new condition where we can see the effects of not having interest management in place. We name the new condition as Auto\IM (“\” represents exclusion).

To summarize, all the versions of the games were still Meerkat games. The conditions only differ in how networking was done:

a. Manual (manual network coding)

b. Auto (fully automated networking)

c. Auto\IM (automated, but no interest management)

d. Auto\IM\DR (automated, but no interest management and no dead-reckoning)

e. Auto\IM\DR\Custom (automated, but no interest management, no dead-reckoning, and no custom sharing using the [Player] attribute)

**Network Latency**

We included two latency conditions to check whether latency had any impact on our measures:

a. Low-latency: Ping time of around 3 milliseconds between the clients and the server.

b. High-latency: Ping time of around 50 milliseconds between the clients and the server.

The low-latency condition was implemented using a dedicated-cable local area network established by a Linksys EZXS88W switch (10/100 Mbps). This condition exhibits typical of a local area network.

We chose 50 milliseconds for the high-latency condition because it is a typical latency in most cases where the game is played in the same continent [21]. The high-latency condition was
realized by applying a simulated latency on the low-latency network. The latency simulation feature is provided by our networking middleware Janus.

All tests were done by running the games for two players (i.e., two clients). We did not vary the number of players in our test conditions. The reason is that an additional player does not have any effect on the performance factors of other existing clients, except for introducing some new entities (e.g., the new player’s avatar). Therefore, from a client’s perspective, increasing the number of remote players is effectively the same as increasing the number of entities in a game. Our different games already cover the variation of number of entities in game.

To summarize our experimental conditions, we had three games from three genres, five levels of networking automation, and two network latency conditions. The measurement metrics and procedure for our experiment are described in the next section.

**5.2.1.2 Metrics and Procedure**

We were interested in measuring performance metrics from two broad perspectives: the game player’s perspective and the system’s perspective.

Our performance measures from the game player’s perspective were: frame-rate, feedback time, and feed-through time. Frame-rate, the number of times the game state is updating/drawing per second, tells whether the game is being drawn fast enough to the player to present a smooth game-rendering. It is also an indicator of the overall game performance; e.g., a good frame-rate implies that the game has access to enough system resources and it is not lagging. The feedback and feed-through times represent the game’s responsiveness to user-input. Feedback time denotes the delay between the occurrence of a user input and the rendering of relevant changes, for example, the time between clicking on a soldier entity and seeing the soldier selected (a ring appears on top of the soldier).
The feed-through time is similar, but it is calculated from a remote client’s perspective. Feed-through time is the time it takes for a client to see the change done in a remote entity, for example, after giving a move command to a soldier, the time it takes for another client to see that the soldier has started moving. It includes the time for one client to update the state, to propagate the information over network, and for the remote client to apply and draw the change.

For our first game (Snowballs), we calculated feedback/feed-through time as the time between the up-arrow-key being pressed (i.e., move forward command) and the aircraft’s new position being drawn. We do this calculation by starting a timer when the input arrives, letting a frame pass in-between, and finally stopping the timer at the end of the next frame when we are sure that the change has been reflected and drawn. For calculating feedback/feed-through time for the RTS game, we consider the time between a move command (right mouse-click) and the soldier’s reaction (start to move). For the racing game, the time between a right-arrow-key press and the car starting to steer right is considered for delay calculations.

In addition to these metrics from the game player’s perspective, we measured system resource utilization factors, such as, CPU, memory, and network usage. We measure CPU usage as the total time (in milliseconds) the game occupied the CPU for completing its execution. We included CPU time spent for game-code execution only, excluding the time spent on system calls. For the memory usage metric, we took the average of working-set physical memory (in bytes) the game used throughout its execution. For network bandwidth usage, we took the average rate of data upload rate and data download rate (separately) for the duration of the game.

The frame-rate, memory usage, network upload rate, and network download rate measurements were sampled every second. We avoided recording them in every frame because that would affect the game’s performance noticeably. We calculated the final average values
using these sampled values. The feedback/feed-through values were, however, recorded each time the relevant user-input occurred. The feedback/feed-through values were also averaged. The total CPU time was measured once at the end of the game.

To ensure that the same gameplay was executed across all our conditions, we instrumented a game recording and play-back system for our games. The games were first played by human players, and we recorded the keystrokes. Depending on the game, the games were played for different durations (Snowballs: 5 minutes, Racing: 3 minutes, RTS: 7 minutes). The games were played long enough to reveal general performance characteristics under our conditions. During our experiments, we played back the input sequences and reproduced the gameplay. First, we carried out the tests in the low-latency condition. Each game was run five times for each of the five automation levels. The same tests were then repeated for the high-latency condition.

We used an Acer Aspire 3820 laptop (Intel Core i5 Processor @2.27 GHz, 4GB RAM) as our server. As clients, we used two Asus G60 laptops (Intel Core i7 Processor @1.67 GHz, 6GB RAM, 1GB nVIDIA GeForce graphics card). As the performance metrics in Meerkat are only relevant on the client-side, we only recorded the values on clients. One of the clients was used to record frame-rate, CPU time, feedback time, upload rate, and download rate. The other client was used to record feed-through time only.
5.2.2 Results

In this section, we present the results we obtained from our tests. We carried out Repeated Measures Analysis of Variance (RM-ANOVA) to determine the effect of five automation levels (described in the previous section) separately on each of our seven performance metrics (also described in the previous section). RM-ANOVA was carried out on the results found from three different games and two network latency conditions. If a statistically significant effect was found using RM-ANOVA, we proceeded to post hoc pair-wise comparisons with Bonferroni adjustment ($\alpha = 0.05$).

The results are presented below, arranged in order of the seven performance metrics we measured (i.e., CPU time, memory usage, network upload rate, network download rate, frame-rate, feedback time, and feed-through time). The purpose of the following subsections is to provide a report of our observed values: the effect found using RM-ANOVA and significance of difference between condition pairs. An interpretation of the findings and a discussion of the results are given later in Section 5.2.3. We have used grouping indicators in our result tables to summarize the significance of differences among condition pairs. Values in the same group denote that we did not find any significant difference for those. Some values have been rounded in the data tables because fractional parts were irrelevant in the context.
Table 1: Total CPU time scores (milliseconds)

<table>
<thead>
<tr>
<th>Snowballs</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>22090</td>
<td>921</td>
</tr>
<tr>
<td>Auto</td>
<td>24926</td>
<td>2117</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>36539</td>
<td>1221</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>59734</td>
<td>1228</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>59698</td>
<td>837</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racing</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>18230</td>
<td>350</td>
</tr>
<tr>
<td>Auto</td>
<td>24879</td>
<td>525</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>25135</td>
<td>561</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>22486</td>
<td>914</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>22249</td>
<td>666</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTS</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>55043</td>
<td>1828</td>
</tr>
<tr>
<td>Auto</td>
<td>67985</td>
<td>1831</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>72200</td>
<td>4078</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>71183</td>
<td>2312</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>71505</td>
<td>1289</td>
</tr>
</tbody>
</table>

**Total CPU time (milliseconds)**

The means and standard deviations for total CPU time scores are presented in Table 1. For the Snowballs game in low latency, the automation level had a statistically significant effect on total CPU time, $F(4, 16) = 836.7, \ p < 0.005$. The mean total CPU time for Manual (values shown in the table) was statistically significantly lower than Auto\IM (p < 0.005), Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). The difference between Manual and Auto was not statistically significant. Auto was significantly lower than Auto\IM (p = 0.011), Auto\IM\DR (p <
AutoIM was significantly lower than AutoIMDR (p < 0.005), and AutoIMDR\Custom (p < 0.005). There was no significant difference between AutoIMDR and AutoIMDR\Custom.

Results for Snowballs in high latency were similar to those in low latency. There was a significant effect of automation, F(4, 16) = 902.8, p < 0.005. This time, the difference between Manual and Auto was significant (p = 0.002). Both Manual and Auto were significantly lower than AutoIM (p < 0.005), AutoIMDR (p < 0.005), and AutoIMDR\Custom (p < 0.005). AutoIM was significantly lower than AutoIMDR (p=0.001), and AutoIMDR\Custom (p < 0.005). There was no significant difference between AutoIMDR and AutoIMDR\Custom.

For the Racing game in low latency, the effect of automation level was significant at F(4, 16) = 120.9, p < 0.005. Manual was significantly lower than Auto (p < 0.005), AutoIM (p < 0.005), AutoIMDR (p = 0.004), and AutoIMDR\Custom (p = 0.002). Auto was significantly higher than AutoIMDR (p = 0.05), and AutoIMDR\Custom (p = 0.004). There was no significant difference between Auto and AutoIM. AutoIM was significantly higher than AutoIMDR\Custom (p = 0.036). The difference between AutoIMDR and AutoIMDR\Custom was not significant.

Similar results were found in the high latency condition for the Racing game. There was a significant effect of automation conditions, F(4, 16) = 166.266, p < 0.005. Manual was significantly lower than Auto (p < 0.005), AutoIM (p < 0.005), AutoIMDR (p = 0.006), and AutoIMDR\Custom (p = 0.002). Auto was significantly higher than AutoIMDR (p = 0.001), and AutoIMDR\Custom (p = 0.021). The difference between Auto and AutoIM was insignificant. AutoIM was significantly higher than AutoIMDR (p = 0.009), and AutoIMDR\Custom (p = 0.002).
The RTS game in low latency showed a significant effect of automation level, $F(4, 16) = 51.5, p < .005$. *Manual* was significantly lower than *Auto* ($p = 0.009$), *Auto\IM* ($p = 0.016$), *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p = 0.001$). However, we did not find any significant difference for other condition pairs.

RTS game in high latency had a significant automation level effect at $F(4, 16) = 28.1, p < 0.005$. Similar to low latency results, *Manual* was significantly lower than *Auto* ($p = 0.049$), *Auto\IM* ($p = 0.006$), *Auto\IM\DR* ($p = 0.03$), and *Auto\IM\DR\Custom* ($p = 0.015$). *Auto\IM\DR* and *Auto\IM\DR\Custom* was significantly different this time ($p = 0.013$). Differences between all other pairs were insignificant.

**Average Memory Usage (bytes)**

The means and standard deviations for average memory usage scores are presented in Table 2. The Snowballs game in low latency condition showed significant effect of automation level, $F(4, 16) = 368.9, p < 0.005$. *Manual* was significantly lower than *Auto* ($p = 0.004$), *Auto\IM* ($p < 0.005$), *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). *Auto* was also significantly lower than *Auto\IM* ($p = 0.002$), *Auto\IM\DR* ($p = 0.004$), and *Auto\IM\DR\Custom* ($p = 0.006$). There was no significant difference among *Auto\IM*, *Auto\IM\DR*, and *Auto\IM\DR\Custom*.

The high latency condition for Snowballs showed the exact results as the low latency condition. The automation effect was significant, $F(2.1, 8.4) = 3003.7, p < 0.005$. *Manual* was significantly lower than *Auto* ($p < 0.005$), *Auto\IM* ($p < 0.005$), *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). *Auto* was also significantly lower than *Auto\IM* ($p < 0.005$), *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). There was no significant difference among *Auto\IM*, *Auto\IM\DR*, and *Auto\IM\DR\Custom*. 

107
Table 2: Average memory usage scores (bytes)

<table>
<thead>
<tr>
<th>Snowballs</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>58908217</td>
<td>125504</td>
</tr>
<tr>
<td>Auto</td>
<td>62181114</td>
<td>612110</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>65243952</td>
<td>149017</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>65441025</td>
<td>79323</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>65664346</td>
<td>346113</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racing</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>54821040</td>
<td>88446</td>
</tr>
<tr>
<td>Auto</td>
<td>58007599</td>
<td>754664</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>57792857</td>
<td>884047</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>57807884</td>
<td>808555</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>57471304</td>
<td>1117356</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTS</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>90670028</td>
<td>1515397</td>
</tr>
<tr>
<td>Auto</td>
<td>112754302</td>
<td>992546</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>112735676</td>
<td>226018</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>112923626</td>
<td>1231732</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>113692586</td>
<td>836046</td>
</tr>
</tbody>
</table>

We did not find any significant effect of automation level for the Racing game (both high latency and low latency conditions).

The RTS game in low latency showed significant effect, F(4, 16) = 483.0, p < 0.005. Manual was significantly lower than Auto (p < 0.005), Auto\IM (p < 0.005), Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). The differences among Auto\IM, Auto\IM\DR, and Auto\IM\DR\Custom were insignificant.
The RTS game in high latency showed the same results. The effect was significant, $F(4, 16) = 43.2, p < 0.005$. \textit{Manual} was significantly different than \textit{Auto} ($p = 0.01$), \textit{Auto\textbackslash IM} ($p = 0.002$), \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p = 0.003$). Other differences were insignificant.

\textbf{Average Upload Rate (kilo-bits per second)}

The means and standard deviations for average network data upload rate are presented in Table 3. Snowballs game in low latency showed a significant effect of automation level, $F(4, 16) = 1809.9, p < 0.005$. \textit{Manual} was significantly lower than \textit{Auto} ($p < 0.005$), \textit{Auto\textbackslash IM} ($p < 0.005$), \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). \textit{Auto} was significantly lower than \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). Difference between \textit{Auto} and \textit{Auto\textbackslash IM} was insignificant. \textit{Auto\textbackslash IM} was significantly lower than \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). There was no significant difference between \textit{Auto\textbackslash IM\textbackslash DR} and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom}.

The Snowballs game in high latency showed exact results as the low latency one. The effect was significant, $F(4, 16) = 23563.6, p < 0.005$. \textit{Manual} was significantly lower than \textit{Auto} ($p < 0.005$), \textit{Auto\textbackslash IM} ($p < 0.005$), \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). \textit{Auto} was significantly lower than \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). \textit{Auto} and \textit{Auto\textbackslash IM} were not significantly different. \textit{Auto\textbackslash IM} was significantly different than \textit{Auto\textbackslash IM\textbackslash DR} ($p < 0.005$), and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} ($p < 0.005$). The difference between \textit{Auto\textbackslash IM\textbackslash DR} and \textit{Auto\textbackslash IM\textbackslash DR\textbackslash Custom} was insignificant.
Table 3: Average network data upload rate scores (kbps)

<table>
<thead>
<tr>
<th>Snowballs</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>26</td>
<td>0.4</td>
</tr>
<tr>
<td>Auto</td>
<td>47</td>
<td>0.3</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>47</td>
<td>0.2</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>64</td>
<td>1.6</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>64</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racing</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>16</td>
<td>0.5</td>
</tr>
<tr>
<td>Auto</td>
<td>28</td>
<td>1.8</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>27</td>
<td>0.5</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>49</td>
<td>0.9</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>48</td>
<td>1.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTS</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>36</td>
<td>1.4</td>
</tr>
<tr>
<td>Auto</td>
<td>58</td>
<td>0.8</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>57</td>
<td>1.7</td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td>150</td>
<td>17.7</td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td>156</td>
<td>2.9</td>
</tr>
</tbody>
</table>

For the Racing game in low latency, the automation level effect was significant at F(4, 16) = 1200.1, p < 0.005. Manual was significantly lower than Auto (p = 0.002), Auto\IM (p < 0.005), Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). Auto was significantly lower than Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). The difference between Auto and Auto\IM was insignificant. Auto\IM was significantly lower than Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). There was no significant difference between Auto\IM\DR and Auto\IM\DR\Custom.
The same results held in the high latency condition for the Racing game. The effect of automation was significant at F(4, 16) = 621.1, p < 0.005. Manual was significantly lower than Auto (p < 0.005), Auto\IM (p < 0.005), Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom. Auto was significantly lower than Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). There was no significant difference between Auto and Auto\IM. Auto\JM was significantly lower than Auto\IM\DR (p < 0.005) and Auto\IM\DR\Custom (p < 0.005). The difference between Auto\IM\DR and Auto\IM\DR\Custom was insignificant.

RTS game in low latency showed significant effect at F(1.074, 4.294) = 240.5, p < 0.005. Manual was significantly lower than Auto (p < 0.005), Auto\IM (p < 0.005), Auto\IM\DR (p = 0.001), and Auto\IM\DR\Custom (p < 0.005). Auto was significantly lower than Auto\IM\DR (p = 0.003), and Auto\IM\DR\Custom (p < 0.005). Auto and Auto\IM were not significantly different. Auto\IM was significantly lower than Auto\IM\DR (p = 0.004), and Auto\IM\DR\Custom (p < 0.005). The difference between Auto\IM\DR and Auto\IM\DR\Custom was insignificant.

RTS game in high latency showed the same effects at F(4, 16) = 1745.0, p < 0.005. Manual was significantly lower than Auto (p = 0.022), Auto\IM (p < 0.005), Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). Auto was significantly lower than Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). The difference between Auto and Auto\IM was not significant. Auto\IM was significantly lower than Auto\IM\DR (p < 0.005), and Auto\IM\DR\Custom (p < 0.005). Auto\IM\DR and Auto\IM\DR\Custom was not significantly different.
Average Download Rate (kilo-bits per second)

The means and standard deviations for average network data download rate are presented in Table 4. The Snowballs game in low latency showed a significant effect at $F(1.3, 5.1) = 2809.8$, $p < 0.005$. Manual was significantly higher than Auto ($p < 0.005$), and Auto\IM ($p < 0.005$); but lower than Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p < 0.005$). Auto was significantly lower than Auto\IM ($p = 0.001$), Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p < 0.005$). Auto\IM was significantly lower than Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p < 0.005$). The difference between Auto\IM\DR and Auto\IM\DR\Custom was insignificant.

The same results were obtained from the Snowballs high latency condition. The effect was significant at $F(1.1, 4.3) = 759.7$, $p < 0.005$. Manual was significantly higher than Auto ($p < 0.005$), and Auto\IM ($p < 0.005$); but lower than Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p < 0.005$). Auto was significantly lower than Auto\IM\DR ($p < 0.005$), Auto\IM\DR\Custom ($p < 0.005$), Auto\IM was significantly lower than Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p < 0.005$). The difference between Auto\IM\DR and Auto\IM\DR\Custom was insignificant.

The Racing game in low latency showed a significant effect at $F(1.1, 4.6) = 84.6$, $p < 0.005$. Manual was significantly lower than Auto ($p < 0.005$), Auto\IM ($p < 0.005$), Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p = 0.006$). Auto was significantly lower than Auto\IM\DR ($p < 0.005$), and Auto\IM\DR\Custom ($p = 0.037$). Auto and Auto\IM were not significantly different. Auto\IM was significantly lower than Auto\IM\DR ($p = 0.001$). Auto\IM\DR and Auto\IM\DR\Custom were not significantly different.
Table 4: Average network download rate scores (kbps)

<table>
<thead>
<tr>
<th></th>
<th>Snowballs</th>
<th>Low Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>High Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td></td>
<td>765</td>
<td>8.9</td>
<td></td>
<td>734</td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td></td>
<td>328</td>
<td>36.1</td>
<td></td>
<td>370</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>Auto\IM</td>
<td></td>
<td>634</td>
<td>2.8</td>
<td></td>
<td>662</td>
<td>5.7</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td></td>
<td>3017</td>
<td>41.8</td>
<td></td>
<td>2723</td>
<td>196.6</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td></td>
<td>3039</td>
<td>117.5</td>
<td></td>
<td>2638</td>
<td>45.3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Racing</th>
<th>Low Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>High Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td></td>
<td>17</td>
<td>0.5</td>
<td></td>
<td>17</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td></td>
<td>34</td>
<td>1.3</td>
<td></td>
<td>37</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Auto\IM</td>
<td></td>
<td>35</td>
<td>1.7</td>
<td></td>
<td>36</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td></td>
<td>55</td>
<td>1.1</td>
<td></td>
<td>58</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td></td>
<td>56</td>
<td>9.0</td>
<td></td>
<td>57</td>
<td>0.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>RTS</th>
<th>Low Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>High Latency</th>
<th>Mean</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td></td>
<td>212</td>
<td>1.2</td>
<td></td>
<td>194</td>
<td>11.6</td>
<td></td>
</tr>
<tr>
<td>Auto</td>
<td></td>
<td>223</td>
<td>1.4</td>
<td></td>
<td>212</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>Auto\IM</td>
<td></td>
<td>222</td>
<td>0.9</td>
<td></td>
<td>213</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR</td>
<td></td>
<td>300</td>
<td>4.1</td>
<td></td>
<td>289</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>Auto\IM\DR\Custom</td>
<td></td>
<td>305</td>
<td>8.8</td>
<td></td>
<td>294</td>
<td>8.9</td>
<td></td>
</tr>
</tbody>
</table>

The Racing game in high latency showed similar results at F(4, 16) = 1640.2, p < 0.005.

*Manual* was significantly lower than *Auto* (p < 0.005), *Auto\IM* (p < 0.005), *Auto\IM\DR* (p < 0.005), and *Auto\IM\DR\Custom* (p < 0.005). *Auto* was significantly lower than *Auto\IM\DR* (p < 0.005), and *Auto\IM\DR\Custom* (p < 0.005). *Auto* and *Auto\IM* were not significantly different. *Auto\IM* was significantly lower than *Auto\IM\DR* (p < 0.005). There was no significant difference between *Auto\IM\DR* and *Auto\IM\DR\Custom*.
The RTS game in low latency showed significant effect of automation, $F(4, 16) = 465.1$, $p < 0.005$. *Manual* was significantly lower than *Auto* ($p = 0.003$), *Auto\IM* ($p = 0.001$), *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). *Auto* was significantly lower than *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). *Auto* and *Auto\IM* were not significantly different. *Auto\IM* was significantly lower than *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p < 0.005$). *Auto\IM\DR* and *Auto\IM\DR\Custom* were not significantly different.

Similar results were found in the RTS high latency condition. The effect of automation level was significant, $F(4, 16) = 244.6$, $p < 0.005$. *Manual* was significantly lower than *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p = 0.001$). *Manual* was not significantly different than *Auto* and *Auto\IM*. *Auto* was significantly lower than *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p = 0.001$). *Auto* and *Auto\IM* were not significantly different. *Auto\IM* was significantly lower than *Auto\IM\DR* ($p < 0.005$), and *Auto\IM\DR\Custom* ($p = 0.001$). There was no significant difference between *Auto\IM\DR* and *Auto\IM\DR\Custom*. 
Average Frame-rate (frames per second)

Table 5: Average frame-rate scores (frames per second)

<table>
<thead>
<tr>
<th>Snowballs FPS</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>59.8</td>
<td>0.05</td>
</tr>
<tr>
<td>Auto</td>
<td>59.9</td>
<td>0.02</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>59.9</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racing</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>59.9</td>
<td>0.04</td>
</tr>
<tr>
<td>Auto</td>
<td>59.9</td>
<td>0.02</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>59.9</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTS</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>58.6</td>
<td>0.35</td>
</tr>
<tr>
<td>Auto</td>
<td>59.0</td>
<td>0.78</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>57.0</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
<tr>
<td>Auto</td>
<td>IM</td>
<td>DR</td>
</tr>
</tbody>
</table>

The means and standard deviations for average frame-rate score are presented in Table 5. We did not find any significant effect of automation level on the frame-rate.
**Average Feedback Time (milliseconds)**

Table 6: Average feedback time scores (milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>Low Latency</th>
<th></th>
<th>High Latency</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Snowballs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>22</td>
<td>0.5</td>
<td>22</td>
<td>0.1</td>
</tr>
<tr>
<td>Auto</td>
<td>22</td>
<td>0.2</td>
<td>23</td>
<td>0.4</td>
</tr>
<tr>
<td>Auto \ IM</td>
<td>23</td>
<td>0.2</td>
<td>23</td>
<td>0.5</td>
</tr>
<tr>
<td>Auto \ IM \ DR</td>
<td>22</td>
<td>0.2</td>
<td>23</td>
<td>0.1</td>
</tr>
<tr>
<td>Auto \ IM \ DR \ Custom</td>
<td>23</td>
<td>0.5</td>
<td>23</td>
<td>0.3</td>
</tr>
<tr>
<td>Racing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>23</td>
<td>0.3</td>
<td>23</td>
<td>0.3</td>
</tr>
<tr>
<td>Auto</td>
<td>23</td>
<td>0.3</td>
<td>23</td>
<td>0.3</td>
</tr>
<tr>
<td>Auto \ IM</td>
<td>23</td>
<td>0.1</td>
<td>23</td>
<td>0.3</td>
</tr>
<tr>
<td>Auto \ IM \ DR</td>
<td>23</td>
<td>0.2</td>
<td>23</td>
<td>0.1</td>
</tr>
<tr>
<td>Auto \ IM \ DR \ Custom</td>
<td>23</td>
<td>0.2</td>
<td>23</td>
<td>0.2</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>30</td>
<td>0.4</td>
<td>32</td>
<td>4.9</td>
</tr>
<tr>
<td>Auto</td>
<td>29</td>
<td>1.5</td>
<td>35</td>
<td>4.4</td>
</tr>
<tr>
<td>Auto \ IM</td>
<td>32</td>
<td>3.8</td>
<td>32</td>
<td>1.0</td>
</tr>
<tr>
<td>Auto \ IM \ DR</td>
<td>29</td>
<td>1.1</td>
<td>31</td>
<td>0.4</td>
</tr>
<tr>
<td>Auto \ IM \ DR \ Custom</td>
<td>30</td>
<td>1.9</td>
<td>32</td>
<td>1.7</td>
</tr>
</tbody>
</table>

The means and standard deviations for average feedback time scores are presented in Table 6. No significant effect was observed on feedback time after our analysis.
Average Feed-through Time (milliseconds)

Table 7: Average feed-through time scores (milliseconds)

<table>
<thead>
<tr>
<th>Snowballs</th>
<th>Low Latency</th>
<th>High Latency</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>63</td>
<td>1.8</td>
<td>110</td>
<td>3.8</td>
</tr>
<tr>
<td>Auto</td>
<td>63</td>
<td>4.1</td>
<td>113</td>
<td>6.5</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>62</td>
<td>2.4</td>
<td>112</td>
<td>7.5</td>
</tr>
<tr>
<td>Auto\IM:DR</td>
<td>64</td>
<td>1.1</td>
<td>115</td>
<td>3.2</td>
</tr>
<tr>
<td>Auto\IM:DR:Custom</td>
<td>61</td>
<td>3.7</td>
<td>113</td>
<td>1.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racing</th>
<th>Low Latency</th>
<th>High Latency</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>62</td>
<td>1.6</td>
<td>111</td>
<td>2.4</td>
</tr>
<tr>
<td>Auto</td>
<td>61</td>
<td>3.8</td>
<td>114</td>
<td>1.7</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>63</td>
<td>1.6</td>
<td>113</td>
<td>2.9</td>
</tr>
<tr>
<td>Auto\IM:DR</td>
<td>61</td>
<td>5.3</td>
<td>112</td>
<td>4.7</td>
</tr>
<tr>
<td>Auto\IM:DR:Custom</td>
<td>60</td>
<td>3.2</td>
<td>110</td>
<td>4.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTS</th>
<th>Low Latency</th>
<th>High Latency</th>
<th>Low Latency</th>
<th>High Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
</tr>
<tr>
<td>Manual</td>
<td>70</td>
<td>2.0</td>
<td>129</td>
<td>8.4</td>
</tr>
<tr>
<td>Auto</td>
<td>66</td>
<td>3.1</td>
<td>134</td>
<td>8.0</td>
</tr>
<tr>
<td>Auto\IM</td>
<td>68</td>
<td>3.5</td>
<td>129</td>
<td>5.5</td>
</tr>
<tr>
<td>Auto\IM:DR</td>
<td>66</td>
<td>1.2</td>
<td>131</td>
<td>4.2</td>
</tr>
<tr>
<td>Auto\IM:DR:Custom</td>
<td>67</td>
<td>1.3</td>
<td>128</td>
<td>5.1</td>
</tr>
</tbody>
</table>

The means and standard deviations for average feed-through time scores are presented in Table 7.

We did not find any significant effect of automation level on feed-through time.
5.2.3 Discussion

In this section, we discuss our experimental results and present generalized conclusions instead of strict statistical analysis (presented in the previous section). Our main goal is to understand the performance overhead of using Meerkat compared to manual networking. A secondary objective is to understand performance implications of different distribution features in Meerkat (i.e., interest management, dead-reckoning, and custom data sharing).

We observed two major patterns in our results from the previous section. First, the findings from the low-latency condition also held for the high-latency tests. For example, the Racing game in low-latency showed that the mean of total CPU time significantly increases from Manual condition (M = 18230 milliseconds) to Auto condition (M = 24879 milliseconds). The same result of significance in difference also holds for the high-latency condition, in fact, with a similar amount of difference (M = 18302 milliseconds to M = 25484 milliseconds). However, there were a few exceptions where the findings were not identical. For example, the RTS game in high-latency did not show a statistically significant difference between average download rate at the Manual condition (M = 194 kbps) and the Auto condition (M = 212 kbps). For all other cases (five of them), we found a significant difference between Manual and Auto. We hypothesize this exception as experimental error that does not prevent us from making a practical conclusion that the values are indeed different. To sum up, we can assume that the findings from high-latency tests are generally the same as those in the low-latency condition. Therefore, for brevity, we will take the findings from the low-latency tests to continue our discussion in the rest of the section.
Another major finding was that there was no significant difference in frame-rate, feedback time, and feed-through time as we moved across different levels of automation. This finding held true for all three games and two latency conditions. Therefore, we will omit these three metrics from our detailed discussion as well.

Finally, we also did not see much significant differences in terms of memory usage. There was a statistically significant difference for some condition pairs. But, generally the differences were of roughly 4MB memory only. As this overhead of 4MB is practically negligible, we will also skip this metric from our discussion.

So, the only metrics that are relevant to our detailed discussion are: CPU time, upload rate, and download rate. We organize the rest of the discussion in two parts. First, we will discuss (Section 5.2.3.1) the performance implication of Meerkat’s major distribution features (i.e., interest management, dead-reckoning, and custom data sharing). Standing on this foundation, we will then discuss (Section 5.2.3.2) how Meerkat generally compared with manual networking.

### 5.2.3.1 Effect of Distribution Features

**Interest Management**
To understand the effect of interest management, we looked at the difference between *Auto\IM* and *Auto* conditions. For example, if we look at the download rate scores for Snowballs low latency case (in Table 4), we find that the download rate at *Auto\IM* (M = 636 kbps) is 94% higher than the previous download rate at *Auto* (M = 328 kbps). It indicates that disabling interest management caused a 94% increase in average download rate for the Snowballs game, which is significant in practical terms. The Snowballs game also showed a significant interest management effect on CPU usage. The total CPU time increased by 47% from *Auto* (M = 24926 milliseconds)
to AutoIM ($M = 36539$ milliseconds). This might be due to spending more CPU behind processing the increased amount of incoming network traffic.

However, we did not find any significant effect of interest management for the other two games. A closer inspection of the games reveals the reasons.

For the Racing game, most of the game entities were static (e.g., trees, road-block and walls) and they were not changing their state (e.g., position) at all. Only the opponent car’s state would change over time. So, there was not a significant change in the amount of incoming network traffic (and its related processing) across two conditions, which may have been the reason for an unchanged download rate and CPU usage.

A similar scenario happened with the RTS game. The number of non-static entities that would change state was very small compared to the total number of entities in the game (e.g., ten opponent soldiers compared to total 500 game entities). Moreover, in the gameplay we recorded, the players were mostly involved in fighting (i.e., they were always in proximity) rather than spreading out in the world. So, almost all remote entities were always inside a client’s interest zone which weakened the effect of interest management (both on download rate and CPU usage).

Therefore, we can conclude that the benefit to be gained from interest management depends on factors of a particular game (e.g., proportion of non-static remote entities, and entity distribution around the game world).

We did not see any effect of interest management on upload rate. This was expected since interest management is only relevant for download data, resulting from state updates of remote entities.
Dead Reckoning

In Snowballs game, we had fifty non-static entities (i.e., snowballs) that were always changing position and angle. Therefore, we expect a significant impact of dead-reckoning for this game as a client would have to deal with much increased incoming state updates if dead-reckoning is disabled. From Table 1, we see a 64% increase in CPU time from Auto\IM (M = 36539 milliseconds) to Auto\IM\DR (M = 59374 milliseconds) after disabling dead-reckoning. The download rate change was even greater at a 375% increase (M = 634 kbps to M = 3017 kbps).

The Racing and RTS games, however, did not have such significant proportion of non-static remote entities (already discussed). Therefore, we did not see any significant change in CPU usage for these games. In fact, we saw a slight decrease in total CPU time, perhaps because the clients no longer had to do extrapolation calculations. For the Snowballs game, this decrease in CPU usage was overshadowed by the greater factor of increased incoming data. Even though not as significant as the Snowballs game, the download rate scores still increased for the Racing and RTS game: 60% (M = 35 kbps to M = 55 kbps) and 35% (M = 222 kbps to M = 300 kbps) respectively.

The upload rates for the Snowballs, Racing, and RTS game increased by 37% (M = 47 kbps to M = 64 kbps), 78% (M = 27 kbps to M = 49 kbps), and 35% (M = 222 kbps to M = 300 kbps) respectively.

Therefore, even though the amount of benefit varied from game to game, dead-reckoning was a much stronger automation feature (than interest management) that improved system resource utilization.
Custom Data Sharing

Custom control of data sharing (by using the [Player] attribute) was a high-level distribution control parameter in Meerkat. We wanted to see whether having access to this parameter made any significant difference.

Surprisingly, we did not find any significant difference after disabling this feature. Perhaps, the reason was that our game entities actually did not have many custom defined fields. Most of the fields that entities needed were already covered by the Meerkat API (e.g., Position, Angle, Hue and Velocity). The only custom fields entities had were several float type fields, used for timing and internal calculations. These fields might not have been enough to demonstrate a significant overhead. If we had enforced the use of some container types as custom fields (e.g., a large List of position values), we might have seen a noticeable difference. But, none of our games needed such large custom data structures.

It might be typical to not see a difference of using custom fields, especially when the API already covers most of the attributes entities usually need. But, it might also be a limitation of our experiment, requiring future study.
5.2.3.2 Performance Overhead of Meerkat

After our discussion of Meerkat’s distribution features in isolation, now we discuss the performance implications of regular Meerkat (full automation), as compared to manual networking. To recall, these two conditions were identified as Auto and Manual in our test results. Our observations are outlined below in terms of different performance metrics.

Total CPU Time
Using Meerkat (Auto) generally implied an overhead on CPU usage across all the games. The increase in total CPU time for the Snowballs, Racing, and RTS games were 13% (M = 22090 milliseconds to M = 24926 milliseconds), 36% (M = 18230 milliseconds to M = 24879 milliseconds), and 24% (M = 55043 milliseconds to M = 67985 milliseconds) respectively.

Average Memory Usage
As we have previously discussed, Meerkat’s overhead on memory usage was practically negligible (roughly 4MB increase in all cases).

Average Upload Rate
The overhead of using Meerkat is perhaps most visible in terms of upload rate. The upload rate increases for Snowballs, Racing, and RTS games were 81% (M = 26 kbps to M = 47 kbps), 77% (M = 16 kbps to M = 28 kbps), and 61% (M = 36 kbps to 58 kbps) respectively. This was not unexpected since the Manual version had in-depth knowledge of game events and state changes and it could broadcast the minimum data when needed. Still, we should keep in mind that this increase will only apply to a small number of game entities, i.e., the avatar entities that a client directly controls. As most games have a single avatar, the upload rate will only be applicable for
one entity. Thus, the total network usage (upstream) will be practically small overall. This is verified by cross-checking the upload rates for our games which are roughly around 50 kbps only (even for the RTS game which had ten avatars).

**Average Download Rate**

From our discussion in Section 5.2.3.1, we have seen that automation features such as interest management and dead-reckoning have a positive impact on average network download. However, we have also discussed why we might not see the benefits in some cases (e.g., in Racing and RTS game) due to factors particular to a game. Our results of download rate illustrate a combined effect of all such factors.

For the Racing game, the *Manual* version was convincingly better than Meerkat where Meerkat implied a 100% increase of average download rate (M = 17 kbps to M = 34 kbps). This is due to the fact that the Racing game could not utilize the benefits of interest management and dead-reckoning at all (described in Section 5.2.3.1). However, the same reasons that made such automation features irrelevant are also the reasons for these games to not have high network traffic. We can verify this from the Racing game where the download rate was only 17 kbps. Therefore, an increase of 17 kbps (total 34 kbps) would still be negligible for practical purposes.

The RTS game scores were pretty even as Meerkat introduced only a 5% increase (M = 212 kbps to M = 223 kbps). This is because the RTS game could utilize the benefits of automation to some extent (discussed in Section 5.2.3.1).

The Snowballs game showed a rather surprising result where Meerkat did significantly better than the *Manual* version. Here, the download rate was decreased by 57% (M = 765 kbps to M = 328 kbps). The Snowballs game was, in fact, the only game where interest management and dead-reckoning was relevant to full extent (Section 5.2.3.1). In this game, we see that automation
features can potentially surpass the benefits of having in-depth game knowledge (that the Manual version had).

To summarize, the observed impact of Meerkat’s automation on download rate can be positive (e.g., Snowballs game), neutral (e.g., RTS game), or negative but practically acceptable (e.g., Racing game).

Frame-rate, Feedback time, Feed-through time
As already mentioned, there was no significant overhead observed in terms of frame-rate, feedback time, and feed-through time.

We have now discussed Meerkat’s overhead for all seven performance metrics. It should be re-iterated that the Auto and Manual versions were not identical in features: the Manual version did not have interest management and dead-reckoning. We may feel that a comparison between Auto\IM\DR and Manual would have been fairer. But, we should remember that Meerkat accepted the performance penalties (e.g., of using proxy objects and interceptions) so that it could later utilize such optimizations (e.g., generic interest management). Therefore, using Auto\IM\DR for comparisons would not be fair.

Another option would be to add interest management and dead-reckoning to the Manual version. The Manual version would then potentially be much better than Auto. But again, the whole point of using Meerkat is that all these complex optimizations are part of the toolkit, whereas, in the Manual version, the game developer would have to implement them by hand. Therefore, we consider the existing comparison between Manual and Auto to be fair.
To summarize and conclude our performance evaluation, we did not see any measurable difference in user-perceptible metrics, such as, frame-rate, feedback time, and feed-through time. The impact on memory usage was practically negligible. We did see some overhead in terms of CPU and network bandwidth consumption. We agree that this overhead might become an issue if games are being developed under strict performance requirements, for example, in low-end hardware platforms (e.g., early generation consoles) or with a limitation of network bandwidth (e.g., overly limited data-plan in a smartphone). Other examples may be highly customized/optimized commercial games where the game developers may need full control over the distribution. A system similar to Meerkat may or may not be suitable for such cases.

However, for other general purposes, we have concluded that Meerkat’s overhead is not prohibitively high. Overall, we have established that the Meerkat model can be implemented in a practical game development system for developing diverse kind of games. Meerkat might imply loss of detailed control or some performance overhead. But, in exchange, the developers get a completely automated game distribution system out-of-the-box. Therefore, our findings with Meerkat provide strong indications that pure entity-based systems can be practical solutions for developing distributed multiplayer games.
5.3 Limitations and Future Work

Throughout the thesis, we have mentioned the limitations and scope of our work as pertinent to different chapters. In this section, we present some more limitations of our work that can be addressed in future:

**Interest Management Policy**

For simplicity of implementation, we have used a simple position based interest management policy for our current implementation (Chapter 4). This policy poses some problems for describing entities that require access to the whole game world. For example, in the Snowballs game, we wanted to add a feature of calculating the winner of the game. For the purpose, we needed a global Referee entity that could access all snowballs in the game world to determine which player has grabbed the most number of snowballs. If interest management was applied for this Referee entity, the entity would never be able to access snowballs from all parts of the world.

We overcame this by having a special attribute for the entities called *IsEverywhere*. When set to *true*, this would indicate that entity does not really have a fixed position and can be considered to be present everywhere. It is effectively an exception tailored for the server that computes interest management. Even though it is an exception, it still makes sense at the entity level. Conceptually, the Referee entity should indeed be able to “see” the whole game state, which we specify by using *IsEverywhere*.

An alternative solution can be a global query mechanism for the clients. The server should host such a query service that can return answers to client queries about the global state of the game world.
**Entity Garbage Collection**

Currently, once an entity goes outside a client’s interest zone, the client continues to keep a lightweight proxy object for serving possible future accesses (Chapter 4). It is possible that the list of such proxy objects keeps growing over a long period of time. This overhead might be negligible for most small or mid-size games that have even several thousand entities. But, other large-scale games may find this overhead significant.

A solution to mitigate this problem might be to use a more complex interest manager that can send commands to permanently destroy entities taking more factors into account, e.g., last access time. For example, if a client has not accessed a hibernating entity for a long period, we may predict that the client is no longer interested in the entity. This is, however, not a full-proof solution as it can potentially break the model if the prediction goes wrong.

Another approach that can be investigated is whether it is possible to maintain one single proxy object instead of separate ones for each hibernating entity. That object may store enough information (e.g., access history) so that the real intended entity can be re-instantiated on demand.

**Consistency Maintenance**

To keep a simple consistency maintenance policy, Meerkat currently does not support concurrent modification of entities by multiple clients. In Chapter 4, we mentioned some approaches that can be integrated to provide support for concurrent entity modifications. Once Meerkat supports such modifications, we may potentially be able to simplify the Meerkat model even further.

Clients may then set property values of non-owned remote entities. We may even be able to get rid of remote method invocations altogether. Each client can then execute method calls on
local copies (of remote entities) and then write the changes back to their Timelines. Overall, the programming model will become more symmetric for owned and non-owned entities.

**Usability Study**

Our attempt to build Meerkat was inspired by the widespread success of other entity-based approaches (Chapter 2). Meerkat extended the entity-based approach to distributed games by providing complete abstraction from network coding. The anecdotal precedents of other similar approaches (Chapter 2) and our experience of working with Meerkat (Section 5.1) suggest that the complete automation provided by Meerkat will only make distribution development simpler. However, more studies may need to be done to focus on the usability aspects of pure entity-based model for developing distributed games.

**Implementation Optimizations**

Currently, we are generating proxy classes and objects dynamically at runtime. We use these proxies for intercepting method invocations at runtime and then to forward them using C# reflection. Static proxies generated using code pre-processing would be more performance efficient. We chose dynamic proxies so that we could use code development supports from existing tools (e.g., static error checking and API hints). Static proxies with development support tools may be investigated in future.

Our current implementation relies on default serialization methods for C#/XNA data structures. This might not be the most efficient way of serializing data, especially for container types (e.g., List, Dictionary). Instead of serializing the whole container on each change, some form of delta-encoding may be applied to communicate only the elements that changed.
It should be re-emphasized that limitations of Meerkat are mostly applicable to its current implementation. Throughout the thesis we have discussed how each of the limitations may be addressed in future implementations. We do not consider these limitations as critical because Meerkat’s purpose was to give only a proof of concept that development and use of such systems is practical.

5.4 Conclusion

In this chapter, we have outlined and described our evaluation of Meerkat. We have summarized how Meerkat proves the practicality of implementing a pure entity-based model for distributed games. We have presented the results of our experiments that reveal the performance overhead characteristics of Meerkat and its automation features. From the results we have concluded that except for games with the most exigent performance requirements, pure entity-based systems can be used for developing networked games in general. Finally, we have presented some limitations and possible future work for Meerkat.
Chapter 6

Conclusion

In this thesis, we have presented an entity-based model for developing distributed multiplayer games. By analyzing contemporary entity-based tools, we have identified the need and lack of a pure entity-based system for networked multiplayer games (Chapter 2).

In Chapter 3, we presented our solution by describing its features for developing a complete game. We also showed how our model seamlessly extends to operation in a distributed context without requiring the game developer to write any networking code.

To show the practicality and soundness of our model, we implemented a toolkit called Meerkat. We have discussed how a pure entity-based system such as Meerkat can be implemented to support networked games (Chapter 4). We have identified some key challenges of implementing such systems (e.g., entity-referencing in the presence of automatic interest management) and presented our solutions. We have also discussed how we integrated several major optimization techniques into our system (e.g., interest management, local-lag, delta-encoding, and selective data serialization). While the current toolkit is aimed at non-expert game developers (e.g., hobbyists) who are willing to give up control for simplicity, the techniques used by Meerkat might be applicable elsewhere (e.g., development of large-scale commercial games).
To evaluate the performance of our system, we built three multiplayer games from different genres: casual action, racing, and real time strategy. We developed different versions of the games to measure the performance implications of Meerkat and its major distribution automation features.

We found that the benefits of automation features (e.g., interest management and dead-reckoning) depend on the nature and design of the games. We presented a detailed discussion of these findings in Chapter 5.

Our primary focus of performance tests was assessing the costs of using Meerkat compared to manual networking. We found that fully-automated networking provided by Meerkat had no impact on frame-rate, feedback time, and feed-through time. However, the cost was noticeable from a system-resources perspective. The most significant was the overhead on network bandwidth usage. In some cases, using Meerkat nearly doubled the bandwidth consumption. However, in some other cases, Meerkat’s bandwidth consumption was surprisingly lower than the manual networking version. Overall, we concluded that Meerkat will be suitable for most cases where we can afford to give up some performance for the simplicity of development.

Although Meerkat has some limitations (e.g., lack of a global-state query mechanism and entity garbage-collector), they are particular to our current implementation and are not critical to the validity of our findings. We have given possible directions for overcoming the limitations and for improving future implementations of similar systems (Chapter 5).
Meerkat was inspired by the anecdotal success of many other entity-based tools that have simplified game development activities. Meerkat shows that it is possible to extend such entity-based models to networked games without revealing underlying distribution technicalities to the game developer.

We believe that such pure entity-based systems will have a large potential impact on game development. Even developers having no prior knowledge of game distribution may develop networked games using such tools. Starting from the basic networking, these tools can automate even complex distribution optimization techniques (e.g., local-lag and delta-encoding), as we did in Meerkat. Thus, developers (experts and non-experts) may spend more time coding the actual game rather than distribution features.

Our implementation of Meerkat and results from our experiments demonstrate that pure entity-based systems are both practical and promising solutions for the development of distributed multiplayer games.
Bibliography


[52] “UDN - Three - GameStateReplication.” [Online]. Available:


