

Firestore – A next generation gaming platform

Scope	3
Rationale	4
Executive Summary	5
System Overview	6
Platform Architecture	7
Modular Service Oriented Architecture.....	7
Event Propagation.....	9
Data Replication.....	9
Principle of Locality.....	10
Modular Game Design	11
Single Thread of Control.....	11
Lobby Data Model.....	11
Performance	12
Summary	13

Scope

This document aims to provide a technical overview of what Firebase is and what it has to offer. This document is targeted towards any interested parties with technical knowledge of modern gaming systems.

Rationale

Online gaming and gambling is a young industry. Much of the software driving the player base is still ad-hoc legacy solutions and much of the new production is kept in-house. However, as the industry matures, enterprise solutions and proven platforms will become more and more important.

With the emergence of a mature industry, time-to-market becomes a very important factor as well as enterprise solutions and services. Operators will find themselves consolidating their platforms, and providers will have a shorter window of opportunity in which to complete and launch their product.

Online gambling has seen a tremendous growth during the last years. The growth of the universal player base together with continuing consolidation of the market will create an even larger player base per network and provider. As the player bases are growing, players are also expecting a more mature and stable service. Downtime and denial of service due to performance issues are an increasing cost in many systems with an increasing player base.

Many operators or network providers struggle with legacy solutions that are not built to handle the non-functional requirements of today such as scalability and high availability. As a result, they are facing major redesigns of their systems in order to handle the paradigm shift of moving from a legacy operation to an enterprise-level scaled business model. Such a redesign represents a very high-risk investment.

Executive Summary

Firebase provides a next generation platform for developing and deploying multiplayer games for gaming or gambling networks with a high number of concurrent users and with high availability.

Firebase is game-agnostic and can be used to implement any game. It provides a scalable general-purpose basis for developing and running games.

Below are some of the traditional major risks associated with developing a multiplayer environment in-house, all of which are addressed by Firebase:

Failure to launch

Of more than 100 massive multiplayer games announced two years ago, fewer than 10% have actually come to market. Many game projects fail in the server development phase as concurrent distributed systems remains complex to implement.

Technical and market risk

Some games that do launch are market failures due to technical problems. They may not scale to large numbers of users; they may be frequently unavailable due to lack of fault tolerance in the underlying infrastructure; or they may be plagued by excessive customer service issues.

Subscriber loss

Gamers are notoriously impatient. Sluggish game play, buggy servers, or poor quality of service leads to boredom or frustration, which translates – often instantly – into a dissatisfied customer and a permanently lost subscriber.

By developing your games using Firebase you minimize these risks and focus on your core business of providing a fun and involving game to the players.

Additionally you would also benefit from these key features when using Firebase:

Generic platform.

Use for poker today, add your multiplayer casino games tomorrow. The platform supports propagation of any proprietary protocol.

Separation of concerns.

Let the game developers do what they do best; develop games. Let the platform handle all non-functional issues such as scalability and transactions.

High availability.

The platform has been designed and built from the ground up to be a redundant and distributed system without just handing off the load to the persistence tier. Should a server go down, game play will not be effected.

Fast time to market

By providing a game API that will ensure all non-functional requirements as well as eliminating multi threading issues in your game implementation, time to market for developing a new game will be drastically lowered.

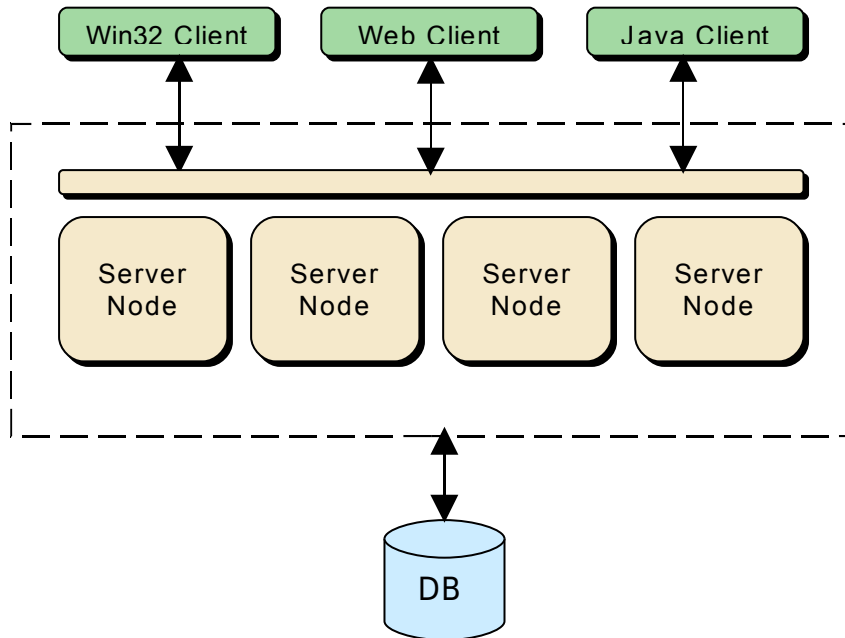
Performance

Firebase is designed from the ground up for gaming, and to be run as a cluster. As such, it can function as a massively concurrent server with low-latency, high throughput and high availability running on standard hardware.

Firebase minimizes risk, reduces time-to-market and provides a scalable platform.

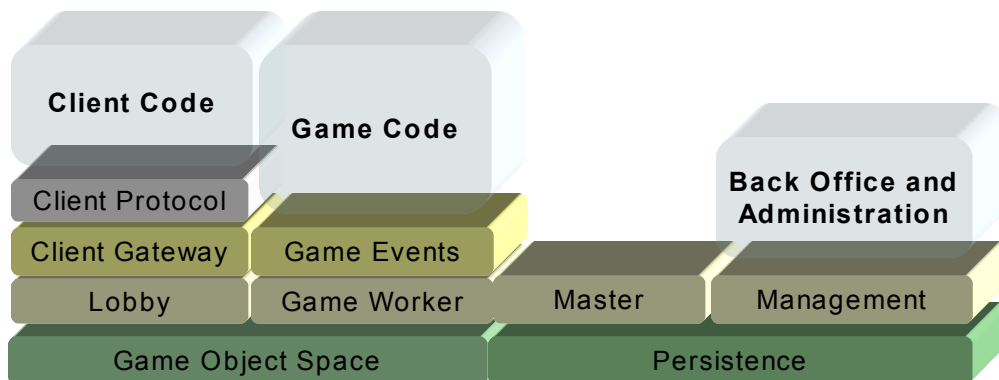
System Overview

Firebase follows the conventional architecture of a 3-tier system defined as client, server and storage tiers, but differs in the fact that the server tier consists of a distributed and dynamic design.



The database is drawn as a single instance, but will generally be a clustered set-up for increased performance and redundancy.

The server tier is built from the start to be a distributed solution. It consists of a service topology of the building blocks shown below.



Blocks with bold text are modules written by the provider and plugged into Firebase. The “client code” consists of the actual game client and uses Firebase communication libraries to connect to the server. The “game code” is the actual game logic, and uses the Firebase API to communicate with the client. The “back office and administration” is usually web services, and they use Firebase management and monitoring hooks and the database connections to provide administration services.

Platform Architecture

A primary driving force behind Firebase is to provide a distributed platform that has high scalability and high availability while retaining enough performance to drive game installations.

Highly scalable systems should be able to smoothly extend its capabilities as the user base grows, in other words: handle more users with added computing resources. High availability systems should suffer from a minimum of down time.

Firestore also provides a uniform view and enables coordination of game-resources in the system (for example, game rooms and tables). Ideally this is kept as a transient system state within the distributed environment.

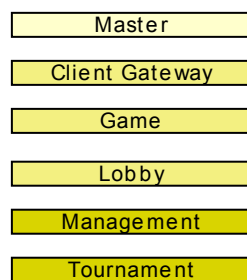
A secondary driving force behind Firestore is to provide a modular design, which is easy and productive to work with for a developer. It should be easy to replace a part within the system as well as extend it with new parts.

Modular Service Oriented Architecture

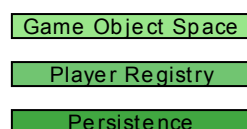
The following are definitions that withhold through the document, but are specifically important for the following section.

- A physical server (or VM) in the Game System is referred to as a Server.
- A service that can be deployed and instantiated multiple times on a Server is referred to as a Node.
- A service that can be deployed and instantiated one time (i.e. single instance) on a Server is referred to as a Service.
- The collection of Services running on a Server is referred to as the Service Stack.
- The collection of Nodes running on a Server is referred to as the Node Stack.

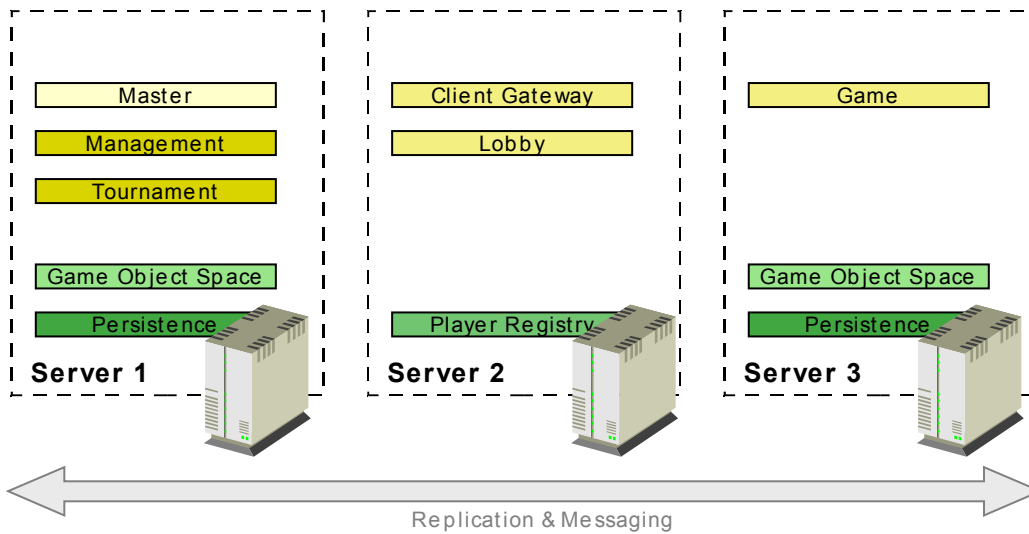
These are the Nodes that make up a complete Node Stack:



The Service Stack is typically fairly large and forms the backbone of the Firestore server. It contains Services for configuration, data sources, entity management and much more. But for this document we will limit ourselves to:

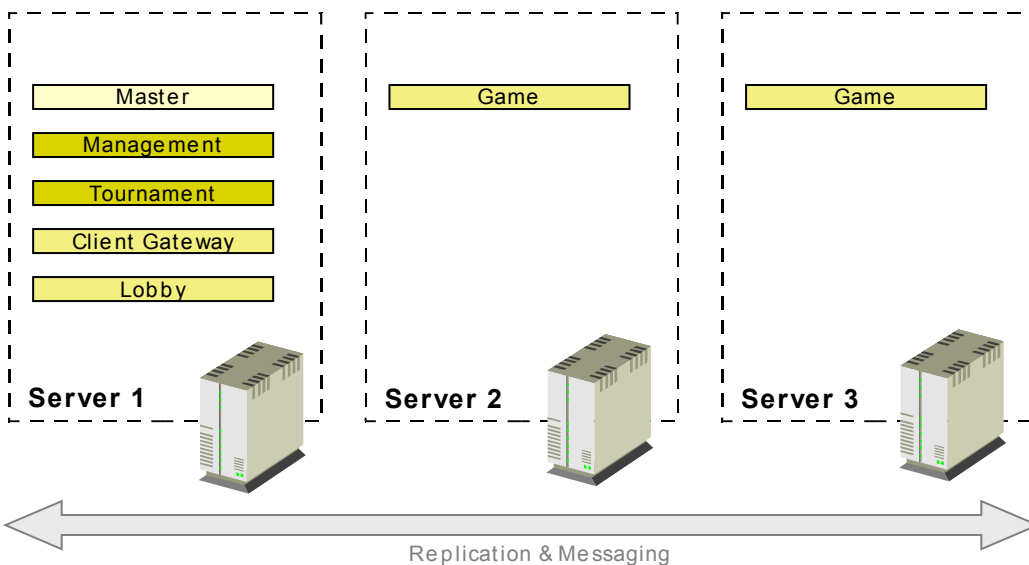


By definition, Nodes can be deployed anywhere any number of times in the system, thus making it possible for a wide range of different cluster topologies. Below is an example of a simple distributed system.



The Services are created dynamically on a need-to-use basis, so the topology definition needs only to specify the Nodes, not the Services.

Nodes deployed on multiple Servers will provide increased capacity as well as redundancy (fail-over). The following topology can thus transparently handle a hardware (or software) failure on Server 2 or 3 without affecting game play.

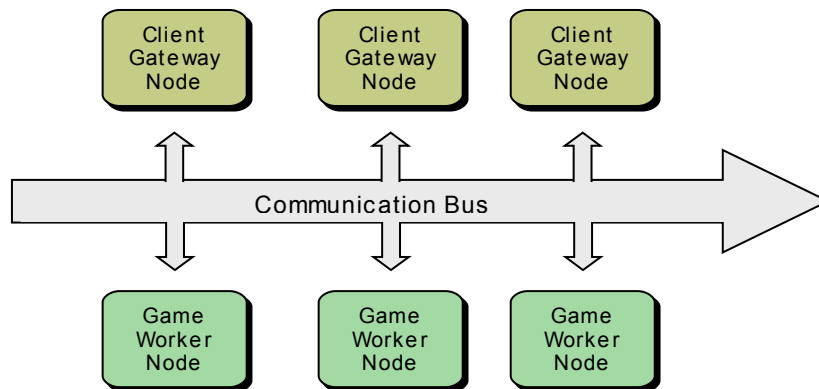


In a full-scale environment you would provide full redundancy for all Nodes as well as head room for fail over. The minimal number of Servers needed for full redundancy is two.

The cluster is controlled by a "master" Node. The master is also the only Node which must exist in a cluster from the start, you can start all other Servers "empty" as long as the master is running. An "empty" Server is running as a container only. Nodes can then be started and stopped, changing the layout in the system dynamically, adapting the system after the current load scenario.

Event Propagation

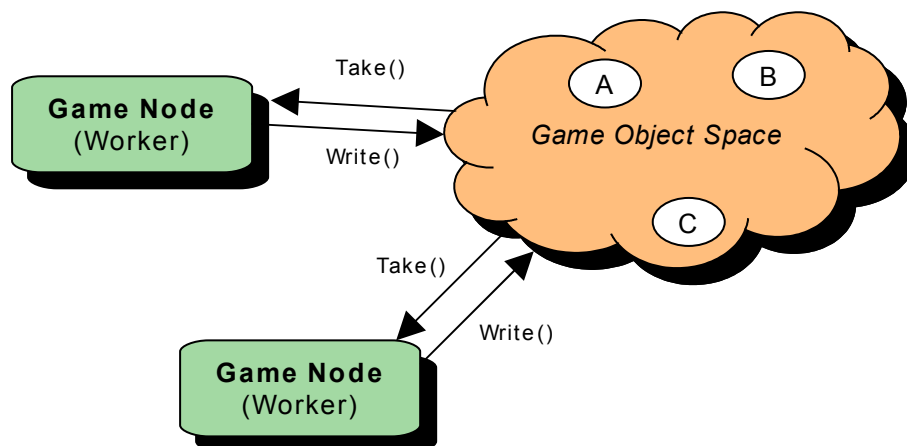
The Communication Bus is responsible for routing all system internal messages. The majority of the messaging consists of game events, generated by the players.



The Communication Bus is driven by a distributed service to provide redundancy. Changes in the cluster layout (for example, dropped or added servers) are detected automatically and the Communication Bus dynamically recalculates a new routing strategy for the topology.

Data Replication

Game objects are data used by the game which need to be kept in memory across the cluster. The Game Object Space (GOS) is a storage area for game objects. Objects in the space can be peeked (read without manipulating them), taken or written.



The worker (Game Node) that needs to perform some work on a game object is usually triggered by a game event from a client, but can also be scheduled events or system maintenance.

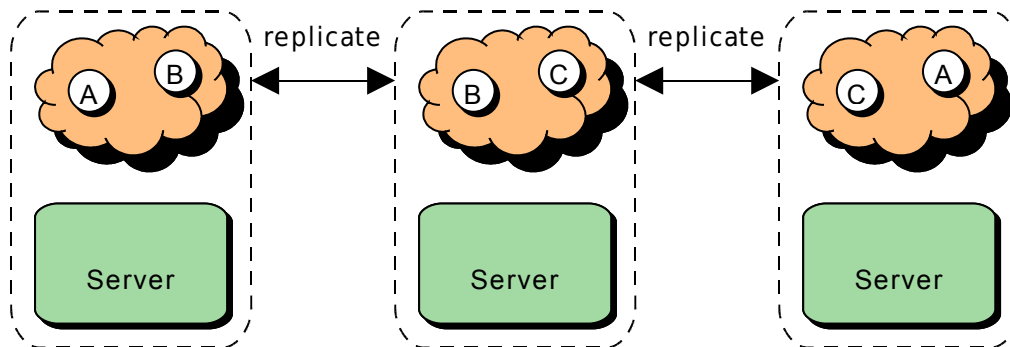
The execution of work on a game object is transactional. Either the full unit of work is executed on the object and the object is written, or nothing is executed at all. There will never be any halfway-executed events.

The workers are agnostic of where the events were originated from as well as what Game Object is designated as the target for the execution.

Principle of Locality

In physics, the principle of locality (POL) is that distant objects cannot have direct influence on one another. In terms of a distributed computer system it means that an object close to the working process takes up lesser resources compared to one which is further away.

In order to comply with the optimal POL, the Firebase replication space can be distributed over the Servers like below.

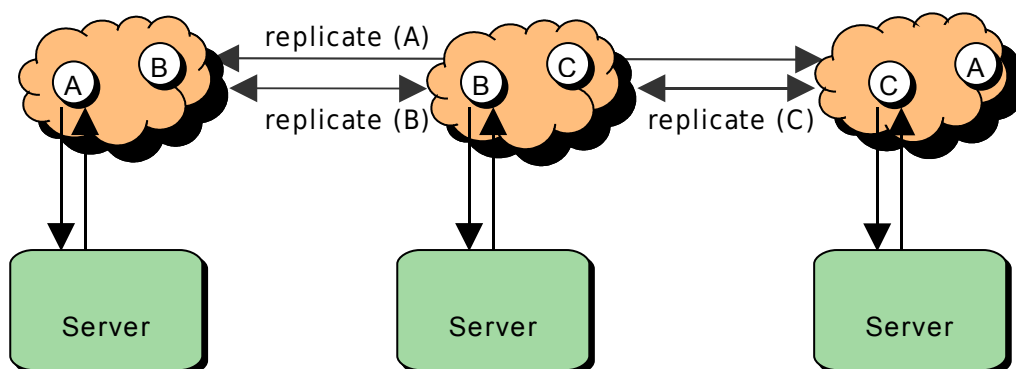


Each Server holds a sub-set of the game objects and not the whole set. This allows for good scalability since we will not run into memory problems due to large object spaces. It also improves performance as it minimizes replication.

The topology allows redundancy and fail-over since we always have a replica of all Game Objects in the space. In practise this is similar to RAID 5 for hard drives, and is sometimes referred to as “buddy replication” or “partitioned cache”.

The physical location (in terms of the actual memory on an actual server) of an object is not static and can change dynamically. More specifically it will change location when a worker asks for the object. This is referred to as “data gravitation”. The Communication Bus will intelligently route events to a Server in order to keep objects as fixed as possible, all in terms with POL.

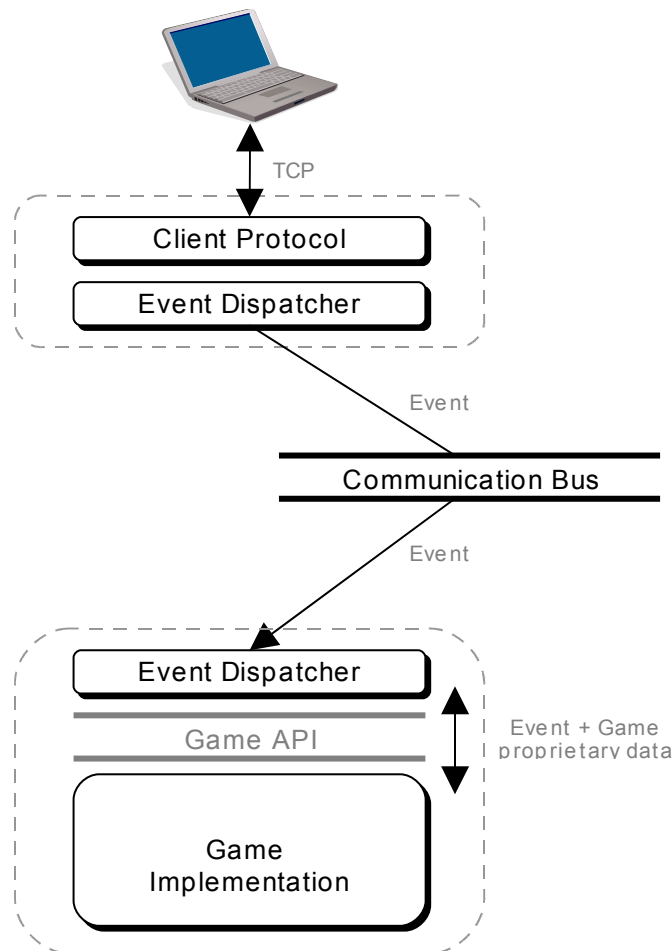
This leads to a working scenario like below:



Using a partitioned replicated space for scaling multiplayer games has also been investigated by Jens Müller and Prof. Sergei Gorlatch from the University of Münster. They refer to this system design as [Multiserver Replication](#).

Modular Game Design

Firestore is designed to handle vastly different games or applications. It handles the communication, scalability and persisting game state, while the Game implementation handles the game logic.



Game specific actions are wrapped in system generic events to keep the platform agnostic to the communication needed for game implementations. It is fully possible to use any proprietary game protocol on the platform.

Single Thread of Control

Although multi-threaded itself, the Game Server API guarantees that there will always be only a single thread executing events in your game implementation. The game developer does not have to take multi threading and concurrency into concern when developing games.

Lobby Data Model

In many legacy systems, distribution of the lobby data uses an unreasonable amount of bandwidth and resources. Firestore keeps a hierarchical data model for lobby data. This data is automatically updated but can also be modified by the Game implementation. The data model is then made available to the client and kept up to date using incremental change notifications.

Performance

Firestore is designed with gaming and gambling in mind. It uses best-of-breed practices and custom made components to minimize latency while still scaling to accommodate a huge number of concurrent players.

The throughput of an installation is largely determined by the size of the Firestore cluster. Firestore is modularized by concerns, making it possible to adapt the system topology according to performance data. To maximize throughput, hardware performance data from the built-in monitoring can be analyzed, and the topology adapted accordingly.

The majority of all latency is currently generated by network transport (TCP) and the actual game implementation. Within Firestore the major latency-critical component is the Communication Bus, which has been custom designed to keep latency to a minimum.

Summary

Firestore is designed to minimize time-to-market, minimize development risk and provide a scalable platform for future development.

By utilizing smart data replication and dynamic cluster topology it provides high availability as well as high scalability. It separates the concerns of the “platform” from the of the “game” and by using it, game developers can concentrate on the game domain, leaving all the other complex issues for Firestore to handle.

Firestore provides an enterprise ready platform to take game development to the next level.