JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

java.sun.com/javaone/sf

Server Architectures for Massively Multiplayer Online Games

Jeffrey Kesselman
Game Server Architect
Game Technology Group
Sun Microsystems, Inc.

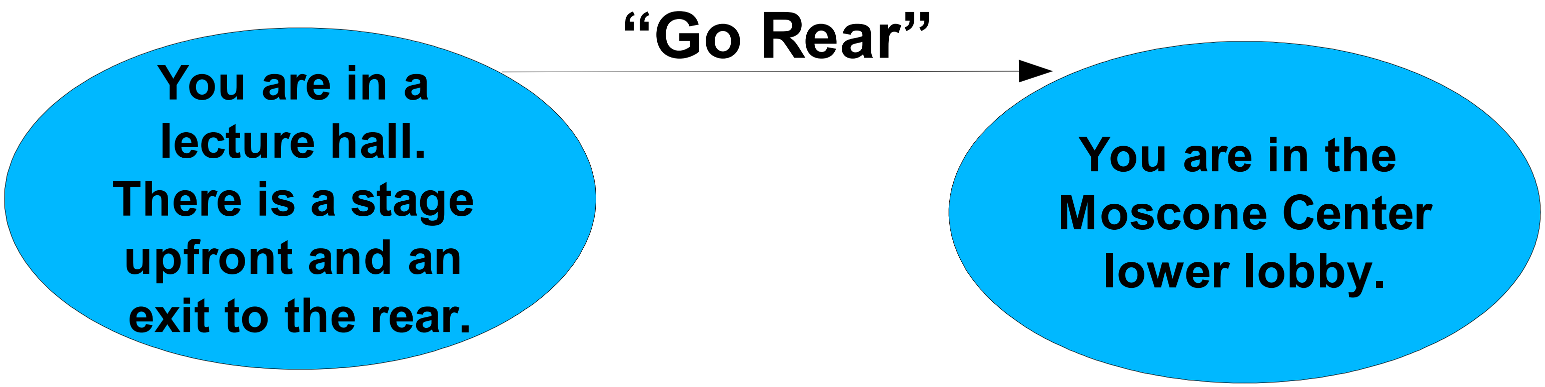
Server Architectures for MMOLGs

What we're going to cover

- A brief history of massively multiplayer online games
- The state of the industry today
- What Sun is bringing to the industry

A Brief History of Massively Multiplayer Games

- MUDs, MUSHs, MUCKs, and MOOs
  - The original “massively” multiplayer online games
    - Online user pop of maybe 50 to 100 people
  - Based on the old Infocom text adventures
    - State machine based textual simulation



“Go Rear”

You are in a lecture hall. There is a stage upfront and an exit to the rear.

You are in the Moscone Center lower lobby.

MUDs, MUSHs, MUCKs and MOOs

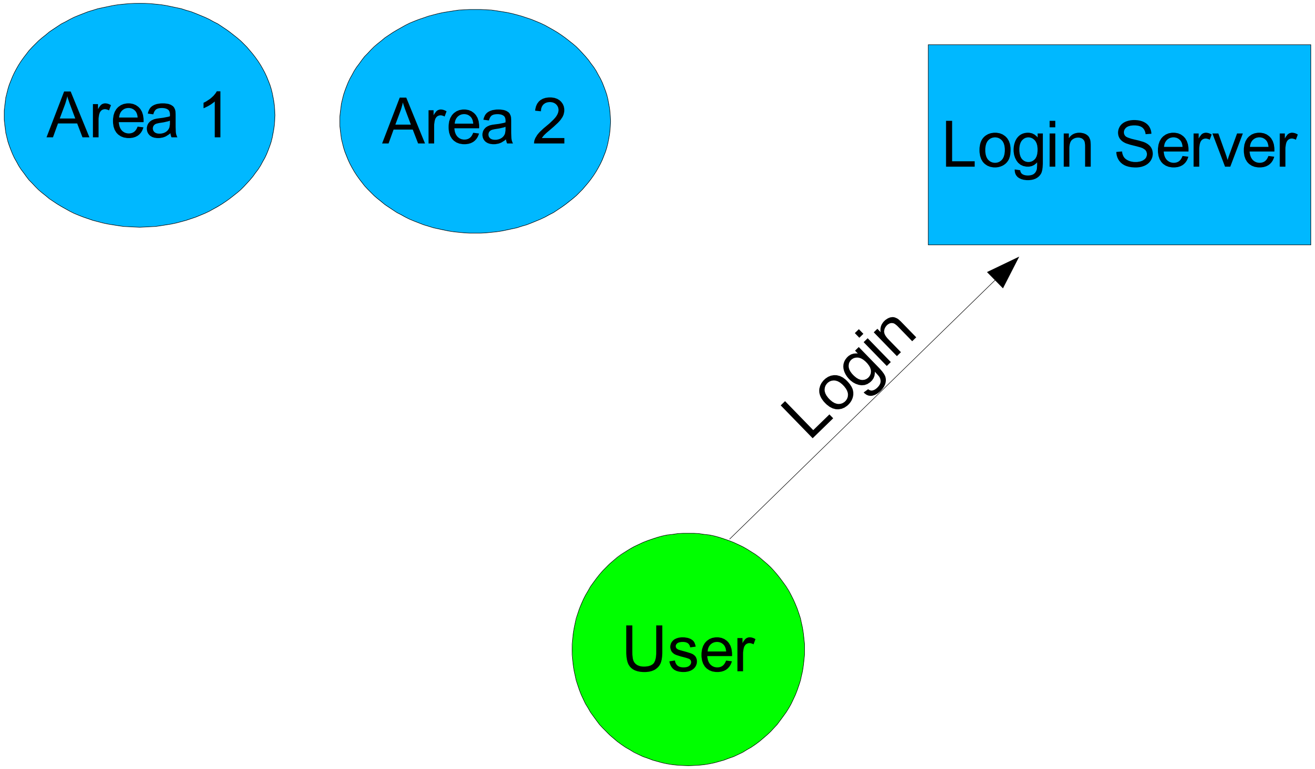
- Still text-based, but with multiple textual clients
- State formalized in concept of “room”
  - Only those in the same room can effect that room or each other
  - N-squared scaling problem solved with simple divide and conquer
- All of world state generally still held in-memory
- Ramifications
  - Limit on world-data size == memory size of host
  - The “fire-marshal” solution for over-crowded rooms

Today: The Everquest Model

- Still room based
  - “Region” or “area” == room + geometry
- Rooms assigned to servers
  - Area Transfer == server switch
- Login Server does initial connection

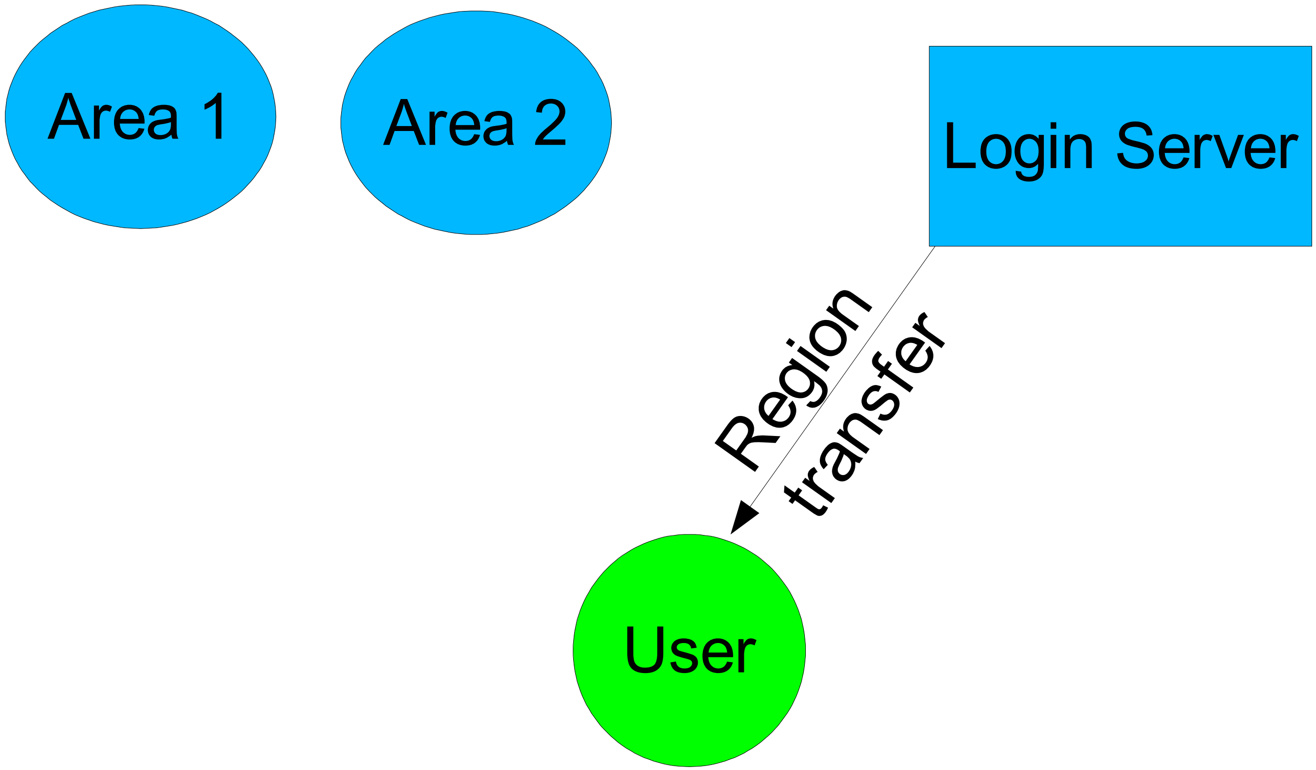
Everquest Login: Connect

- User connects to Login server and authenticates



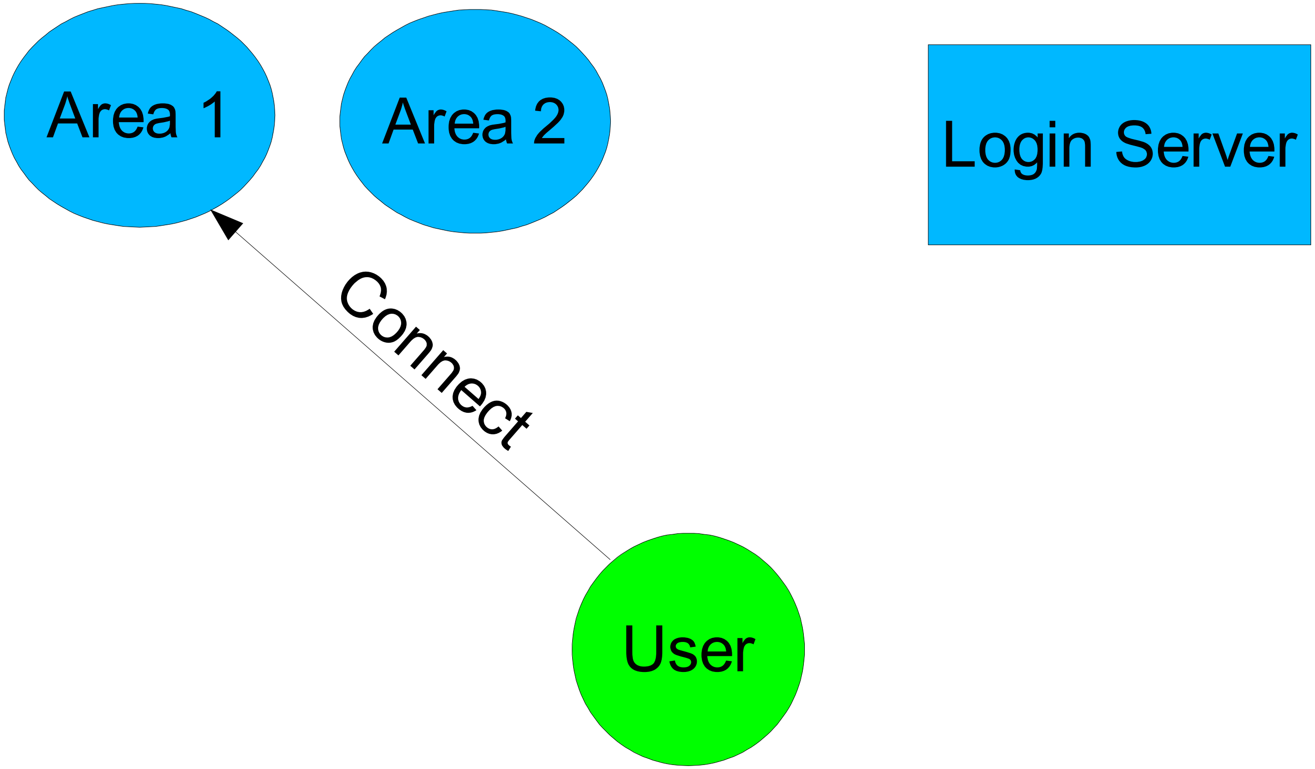
Everquest Login: Initial Region

- Login server transfers user to last region



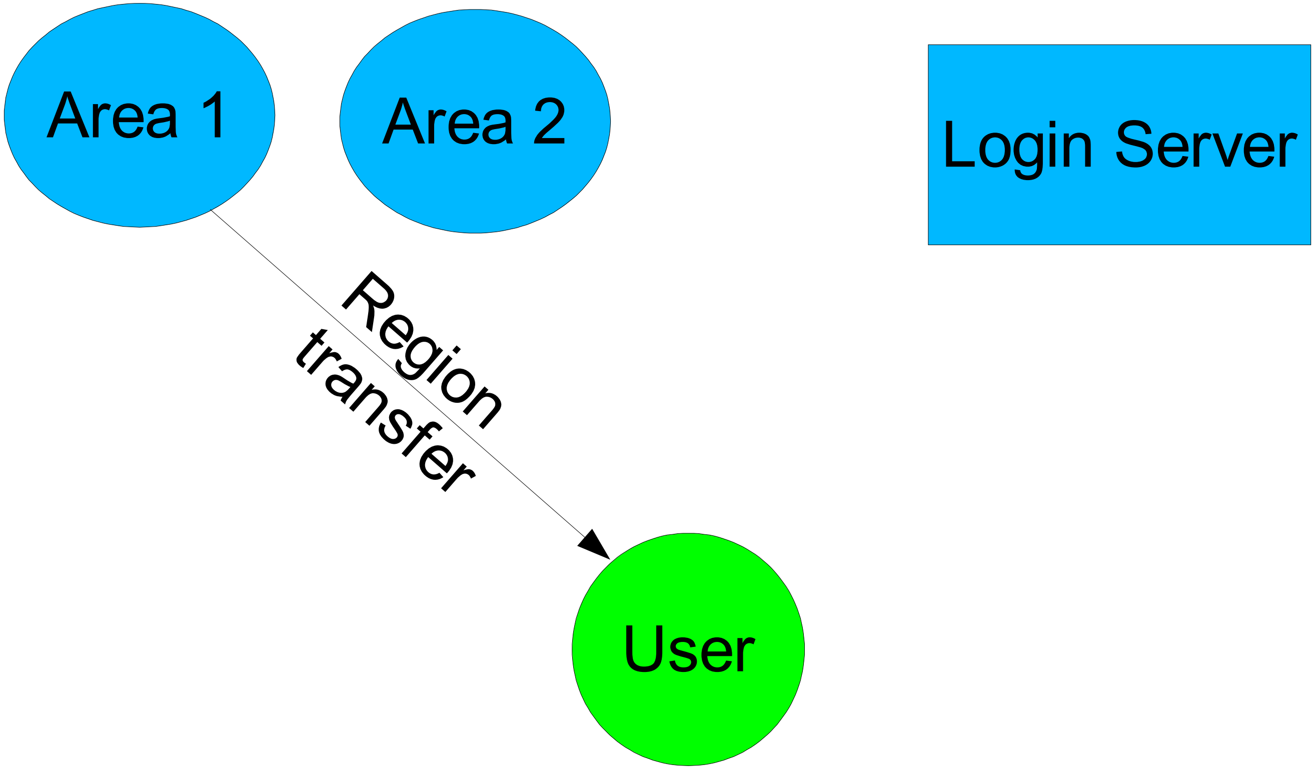
Everquest Login: Initial Region

- User drops connection to login server and connect to Area



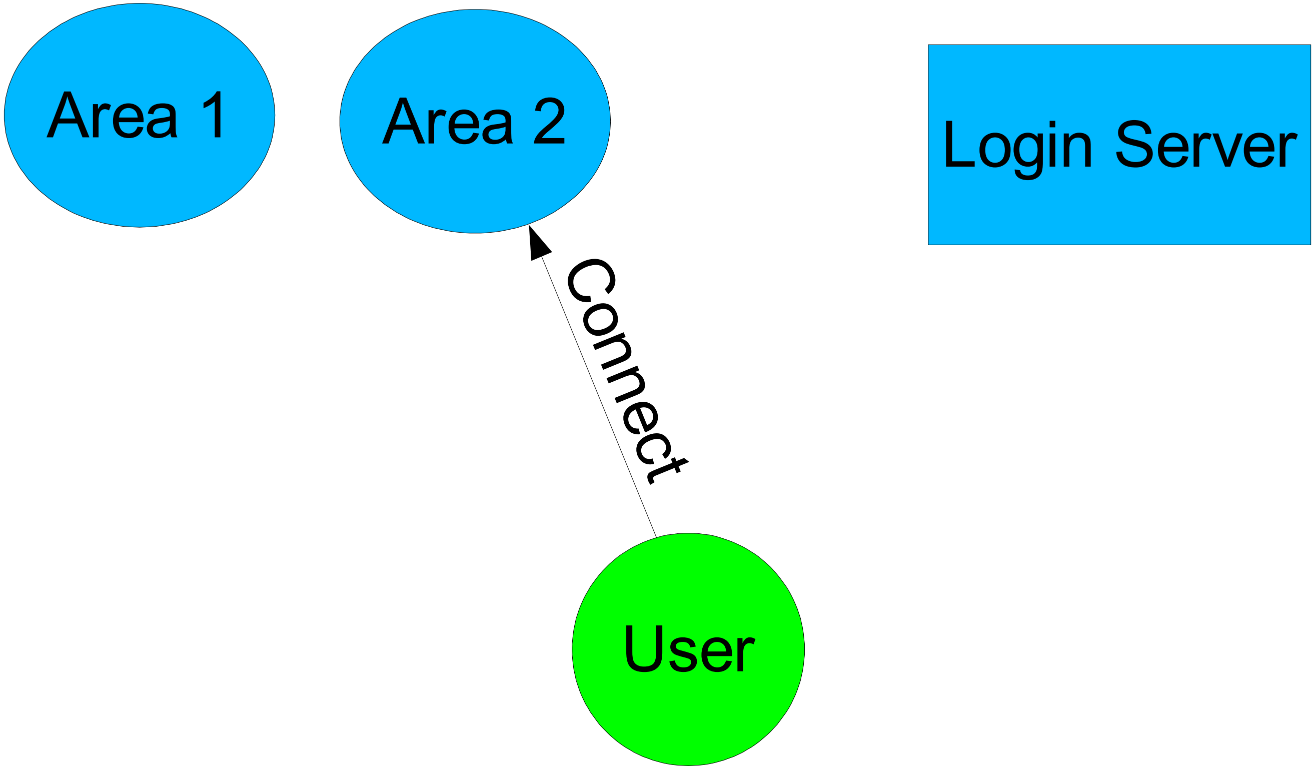
Everquest Login: Initial Region

- When user triggers area transfer, area sends transfer command



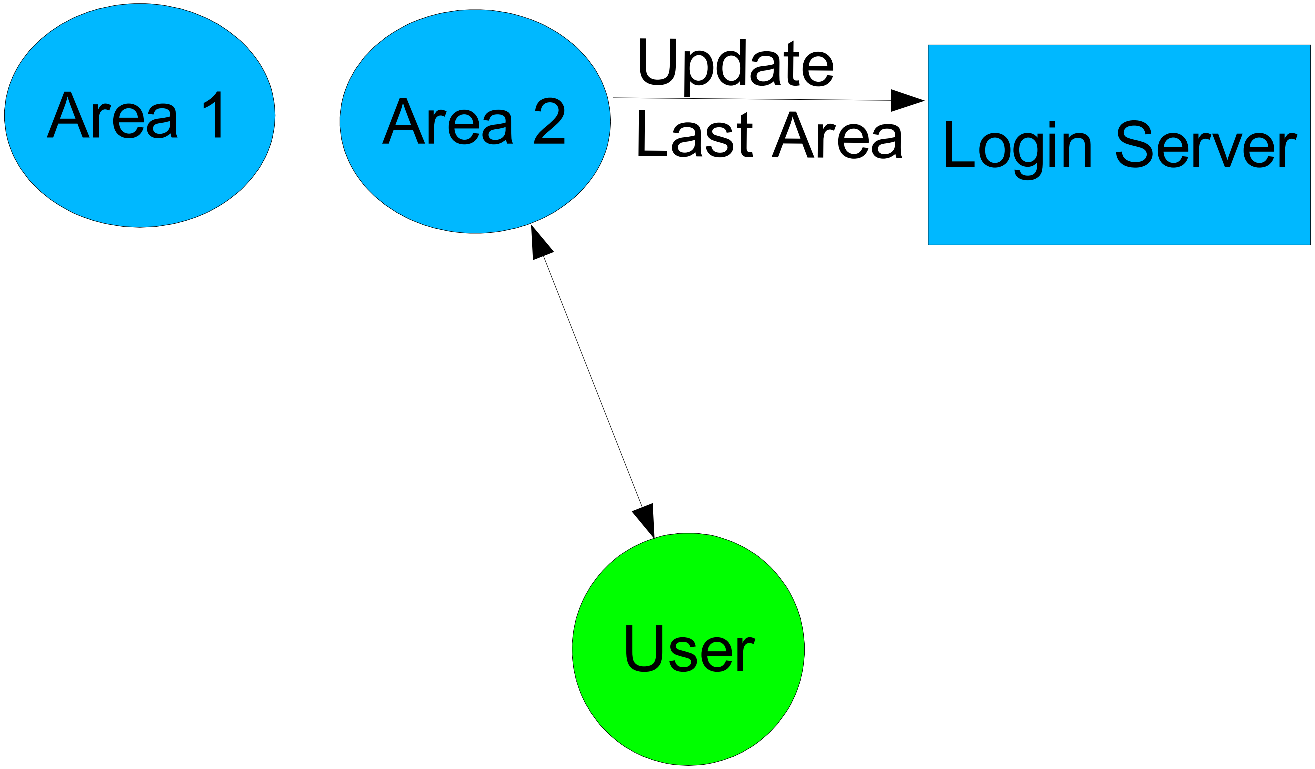
Everquest Login: Initial Region

- User drops connection to Area 1 and connects to Area 2



Everquest Login: Initial Region

- Area 2 updates login server on user's last area



Still a MUD Basically

- “Rooms” (Areas, Regions) organize resource usage
- Room state held in memory
- Now one server per room, which increases capacity

The Limitations of the Model

- This is a perfect model if people spread themselves out naturally in a Gaussian distribution
  - People are social—they clump
- Wasted systems where no one is in that region
  - CPU idle, memory holding state
- Fire marshal limit is bigger but still present
- Not fault tolerant
- Static world due to limited persistence
- Area boundaries unnatural

Sharding

- Solved the fire marshal problem by duplicating entire game—A shard == a duplicate cluster
- Issues:
  - Now you are wasting N servers in light used areas
  - Split the user-base—Bad for business
    - Hurts recruitment
    - Makes it harder to maintain "critical mass" on any one shard

Replication

- Fault tolerance through replication of servers
  - Now wasting 2xN servers for each light used area
  - Double point of failure

Hidden Boundaries

- Hide the boundaries by over-lapping geometry and having “leaky edges”
  - Works pretty well to make world appear continuous
  - This model maps well to a grid
- Issues:
  - Can still get stutters at region transfer time
  - Redundant data means regions have smaller actual play areas (limit is memory size)

Spatial Subdivision

- Solve fire marshal problem by dynamically creating sub-regions
- Issues:
  - Visibility dynamically decreases as regions get sub-divided
  - Still doesn’t solve waste in unused base regions
  - Complicates replication and makes it more expensive
  - Practical limits to sub-dividability (all the players standing right next to each other)
  - Will **not** map to a grid

And So We Come to Today...

- What you have just seen is the limit of MMOLRPG tech deployed today
  - Not fault tolerant
  - Expensive/wasteful
  - Limited persistence
    - Only char data—best systems check point every 15 min
  - Scaling only by splitting user base

The Sun Game Server

- A container system for highly scalable, highly efficient, fault tolerant, dead-lock proof, completely persistent, load-balanced execution of event-driven simulations
  - Boy that's a mouthful, huh?
- Based on a Paradigm Shift
  - Observation 1: Geometry is data
  - Observation 2: Data doesn't use processor
  - Observation 3: Users use processor
  - Conclusion: Assign compute resources to **users,** not regions

What the Developer Sees: GLOs

- Developer creates a world of Game Logic Objects (GLOs)
  - "Real objects"
  - Object register to handle events
- GLO(s) is/are defined by a Java™ Class
  - Must be serializable
  - If it is going to receive an event, must implement the handler interface for that event
- GLO code is apparently mono-threaded
  - No synchronization necessary

What the Developer Sees: Events and Tasks

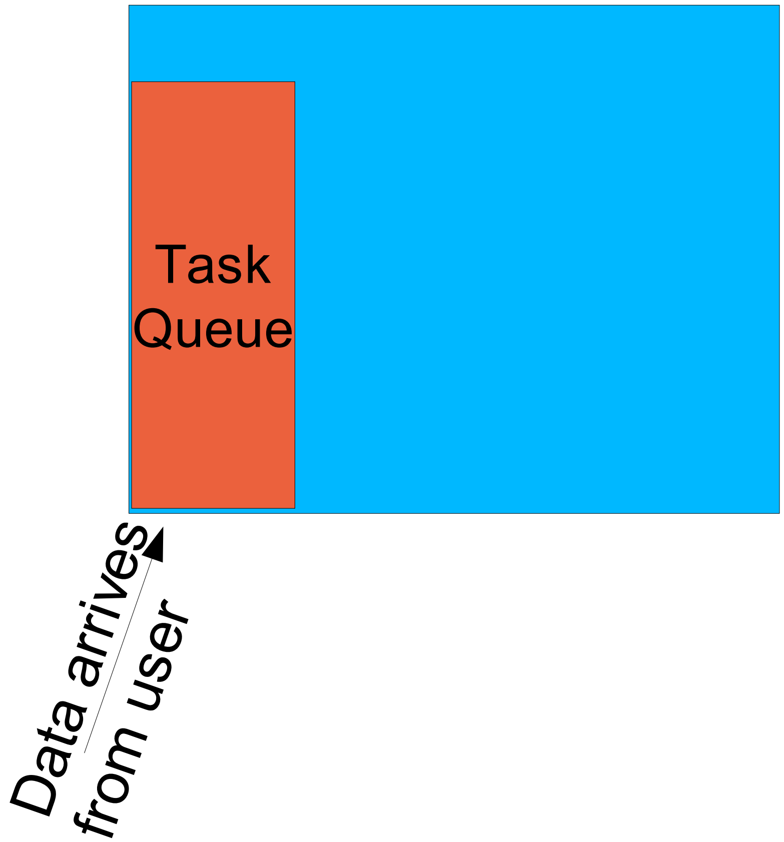
- When an event arrives a Task is queued
  - Task is define as a GLO reference, a method to call, and parameters to pass to it
  - When Task runs the GLO is “woken up” and the method is called
    - GLO can wake up and call other GLOs
    - GLO can register event listeners
    - GLO can queue tasks
    - GLO can execute arbitrary method code
  - Tasks can either commit or abort
    - On commit, modified GLOs are written back and put back “to sleep”
    - On abort, modified GLOs are rolled back and put to sleep. Task is re-queued for later execution

What the Developer Sees: Waking Up a GLO

- A GLO can be woken up by a get() or by a peek()
  - get() is a write lock. GLO is “owned” by that task and will be updated at task completion
  - peek() is a non-repeatable read; a task-local copy of the GLO at its last saved value is created and the GLO is not updated at task completion
  - Initial object fetch in task is always a get()
- get() causes potential contention; for maximal parallel performance an app should use peek() whenever possible
  - Some fine points and “best practices”

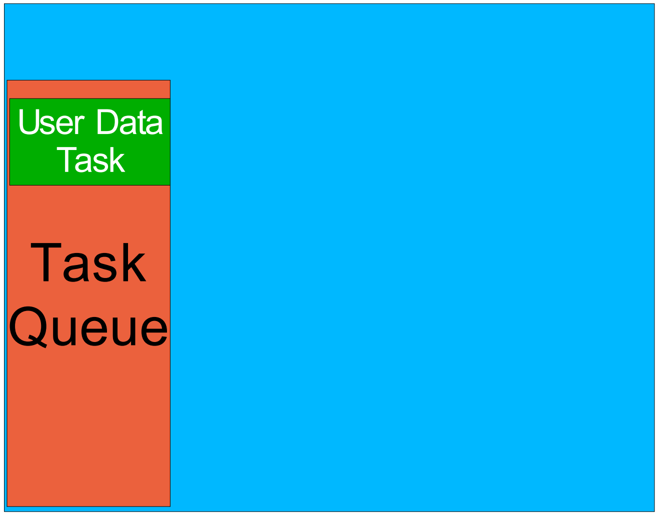
Life Cycle of an Example Event

- Step 1: Packet arrives from user



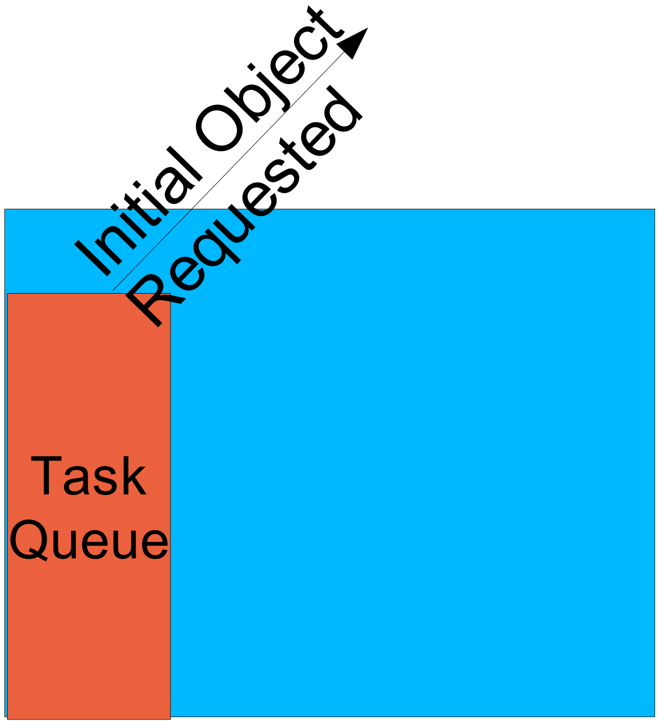
Life Cycle of an Example Event

- Step 2: Event is queued for data packet



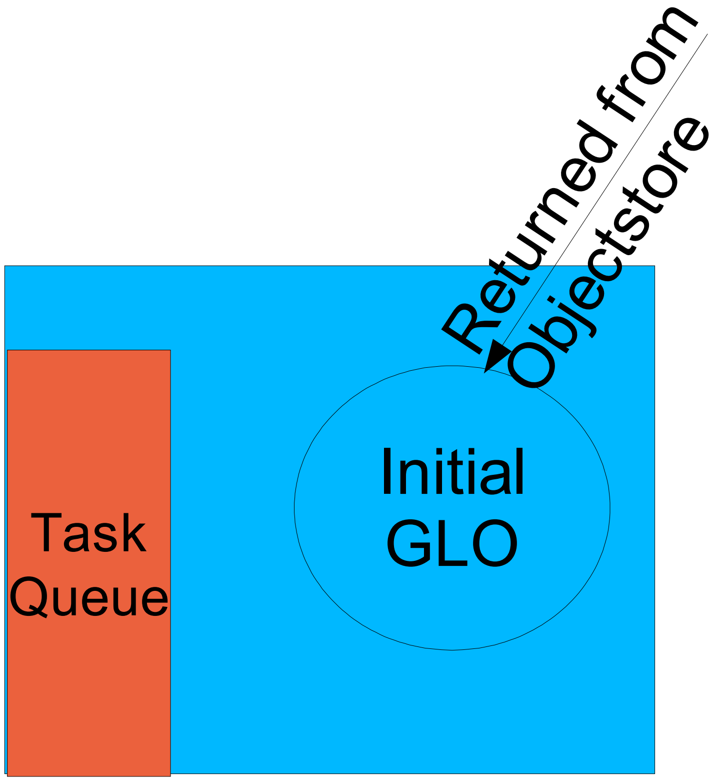
Life Cycle of an Example Event

- Step 3: Event is taken off queue for execution
  Initial object is fetched from Objectstore



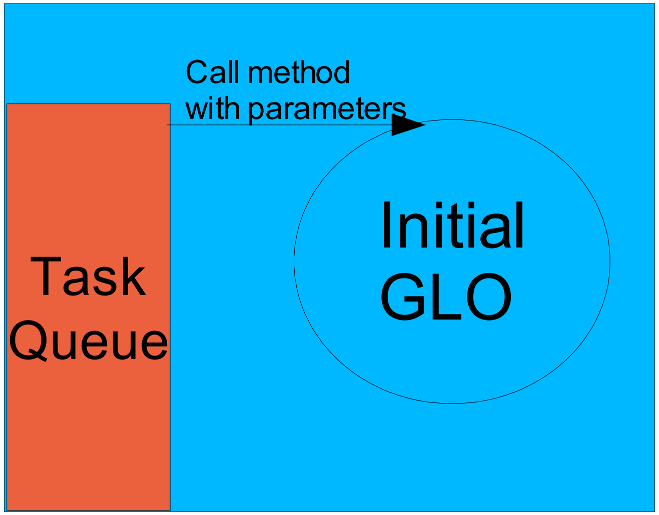
Life Cycle of an Example Event

- Step 4: Object store returns GLO from get()



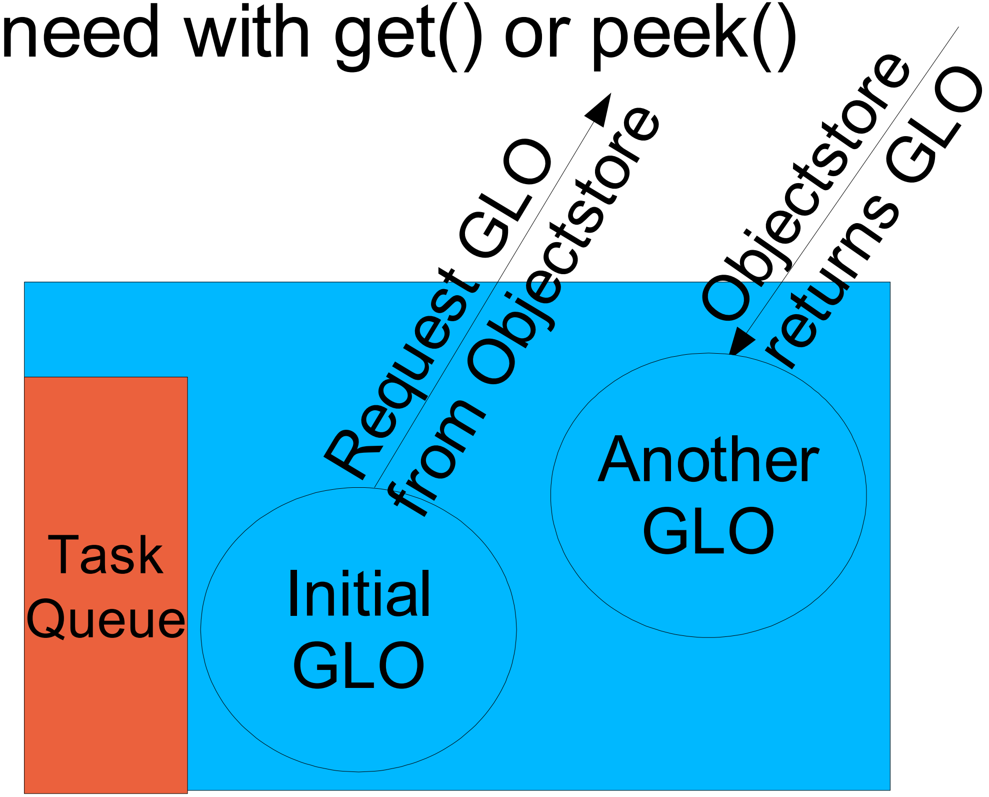
Life Cycle of an Example Event

- Step 5: A transactional context and a thread are assigned to task; initial method invoked on that thread



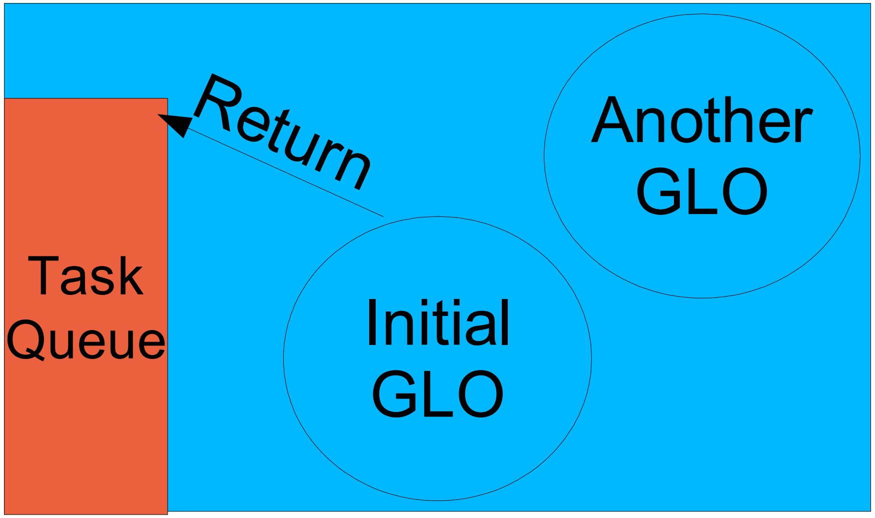
Life Cycle of an Example Event

- Step 5: Initial GLO chains in other GLOs as need with get() or peek()



Life Cycle of an Example Event

- Step 6: Initial GLO returns from initial method call



Life Cycle of an Example Event

- Step 7: Transaction is committed; GIOs return to Objectstore

Life Cycle of an Example Event

- Alternate Step 7: Transaction is aborted; GIOs are rolled back and return to Objectstore

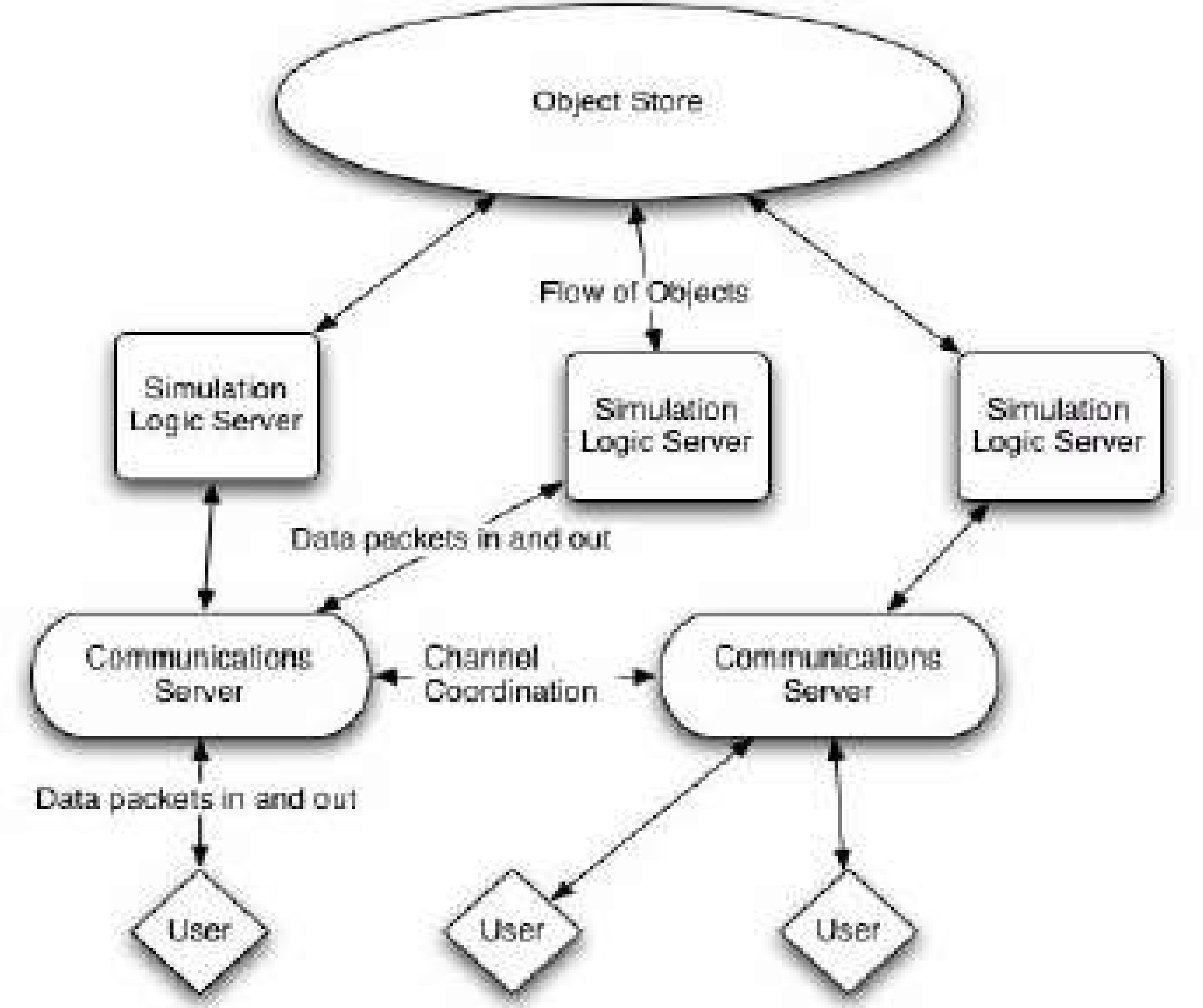
That’s It

- The horizontal scaling, fault-tolerance, persistence, and load balancing are all properties of the underlying system

How the Magic Works!

- Bet you’re dying for the next slides...

The Sun Game Server Architecture



Okay, So What Does That Mean?

- In Architecture it is structured much like a classic 3 tier enterprise system
  - Comm Layer (Edge Layer)
  - Stateless Game Logic Layer (Business Logic)
  - Object Store (Database)
- In implementation it is radically different because its needs are different
  - Total input to output system latencies in 10s of ms
  - Requires no multi-threading knowledge to use
  - Requires no database knowledge to use
  - Handles persistence (almost) orthogonally
  - Very simple coding model

Tier 3: The Objectstore

- This is where a lot of the magic happens
- Looks to tier 2 like a highly available cloud of persistent objects
- All game state for all games running on the Sun's game server back-end infrastructure resides here
- Tier 2 checks objects out as needed by a task and returns them when tasks complete or abort
- It is a **very** fast fault-tolerant horizontally scalable transactional database
  - Time to fetch and de-serialize an object is in the **tenths** of milliseconds

Tier 2: The Game Logic Layer

- This is a stateless task processor
- Users are assigned to one of a horizontally scaled set of stacks
- As user events come in, they create tasks in a task queue
- Because this layer is stateless, if it dies, the users just reconnect to another stack and keep on playing

Tier 2: Task Abort

- Tier 3 (Objectstore) provides deadlock detection through Timestamp Ordering
  - Std deadlock avoidance algorithm
  - “Fair” (ensures no starvation)
- On potential deadlock, tier 2 is notified and newer task aborts
  - GLOs rolled back
  - GLO locks released
  - Task surrenders thread and transaction
  - Task is re-queued for later execution
- Obviously lots of deadlocks hurt efficiency
  - Game Logic Layer tracks and gives warnings

Tier 1: The Comm Layer

- "User manager" is the only part that knows how client is connected
  - Generates user login/logout/packet events
  - Handles server discovery and selection
  - Different user managers for different connection strategies
    - e.g. Gamespy, HTTP, etc.
- Makes the user base seem contiguous
  - Tier 2 just sends messages to user IDs; doesn't care where they are actually connected
- Provides "shortcut" for client to client comm
  - Never goes to tier 2
  - Safer then direct client to client connection

Failure and Recovery Modes

- N horizontal stacks fail
  - Users reconnect to other stacks and go on playing
  - Time to reconnect hidden by client dead-reckoning
  - Performance degrades evenly throughout back-end
  - Only loss is tasks on queue (unimportant)
- Entire back end fails
  - Object store is left in a referentially integral state
  - At most a few moments of gameplay are lost
- Resources added to back end
  - New stack calls out and says stack available
  - Heavily-loaded stacks force some of their user to reconnect

Dynamic Load Balancing

- Stacks send periodic load updates
- Heavily loaded stacks give up users to lightly loaded ones
- One stack can run **many** different games at once
- System load balances across **all** games
  - When games need more of the back end they get it
    - Games that are always lightly used only use that percentage of the resources
    - Makes many lightly used games as economically sound as one heavily used game
    - Makes “pay as you use” models possible

That’s It

- Obviously the scalable Objectstore is a major piece of the technical puzzle
  - Current implementation written ontop of TimesTen
  - Sun-magic to make it horizontally scale for such a write intensive application
- The “app server” for near real-time event-driven simulations
- Currently in ongoing development
- White paper down at the entertainment pavilion

The Rest of the Story

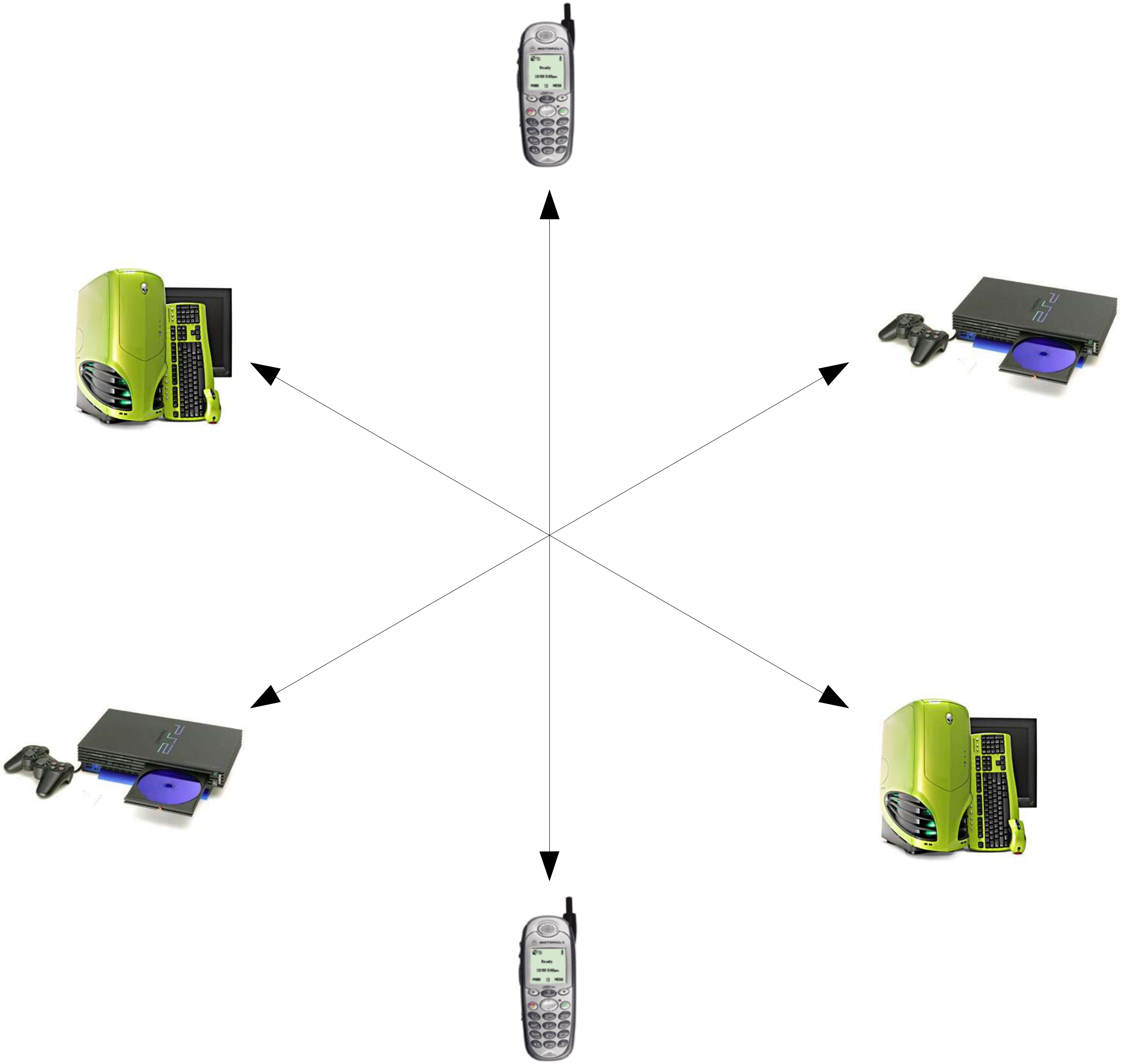
- All that other back-end stuff we do so well...
  - Billing, authentication, portal, etc.
- The Cell Phone infrastructure
  - Provisioning, communications, etc.
  - All the cell phone stuff the User Manager doesn't deal with
- Athomas Goldberg is your man!

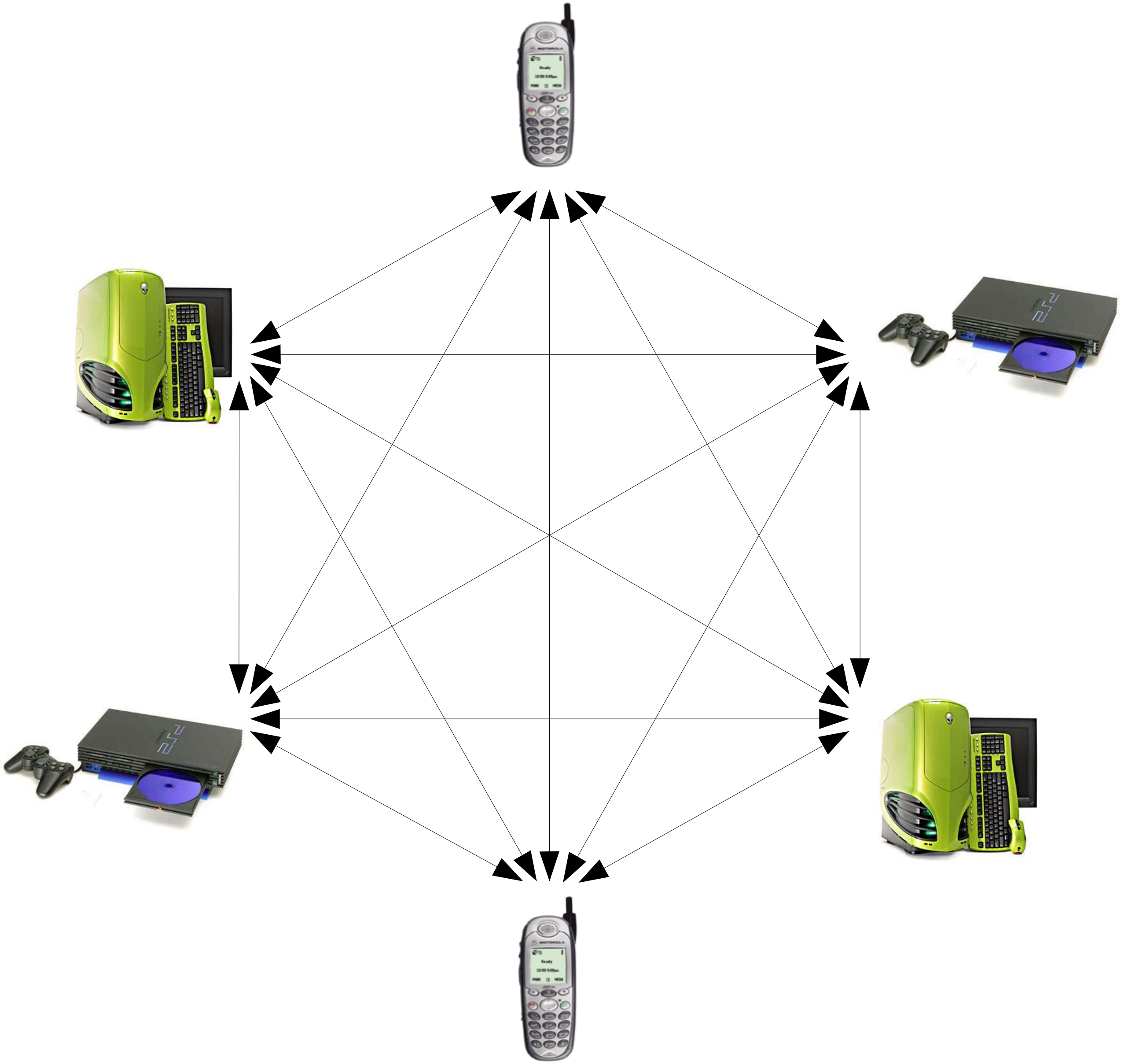
JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Bridging the Network Divide

java.sun.com/javaone/sf





Game Design Challenges

- Mobile devices are currently treated like consoles
  - Need to find activities appropriate to each device
- Need to look for unique games
  - Games that use other parts of the device
    - Camera, GPS, Controllers, etc.
  - Massively Multiplayer Casual Online Games
- Mobile devices are social devices
  - Leverage social interaction between players
  - Portals into a shared experience

Technical Challenges

- Need common infrastructure
  - Cost of developing online capability for a single platform can crush a game
- Need to standardize, so game developers can focus on games
  - Cross-network communication
  - Federated identity
  - Micropayments and billing
- Need to work with developers and operators
  - Develop solutions to address the entire delivery chain

Business Challenges

- Pricing models
  - Existing models don’t support multiplayer
    - “Mobile Everquest” would cost you $10,000/month
- Distribution and management issues
  - Retail vs. on-demand distribution
  - Delivering a service vs. delivering a product
- Hosting issues
  - Carriers don’t trust game code on their networks
  - Demand high quality of service from game hosts
- Need to resolve cross-carrier gaming
  - Carriers resist
  - Developers (and customers) insist

JavaOne℠

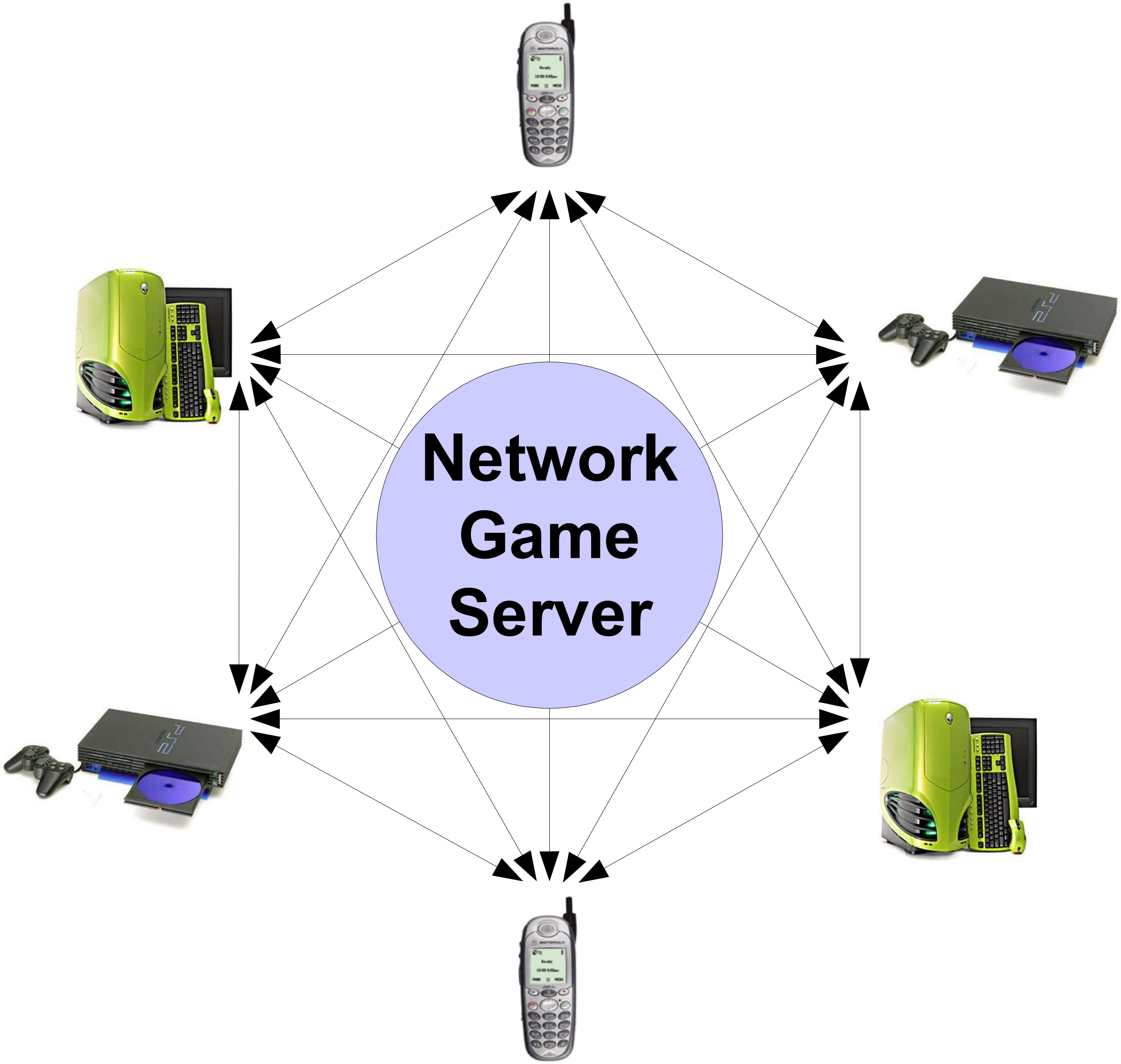
Sun's 2004 Worldwide Java Developer Conference℠

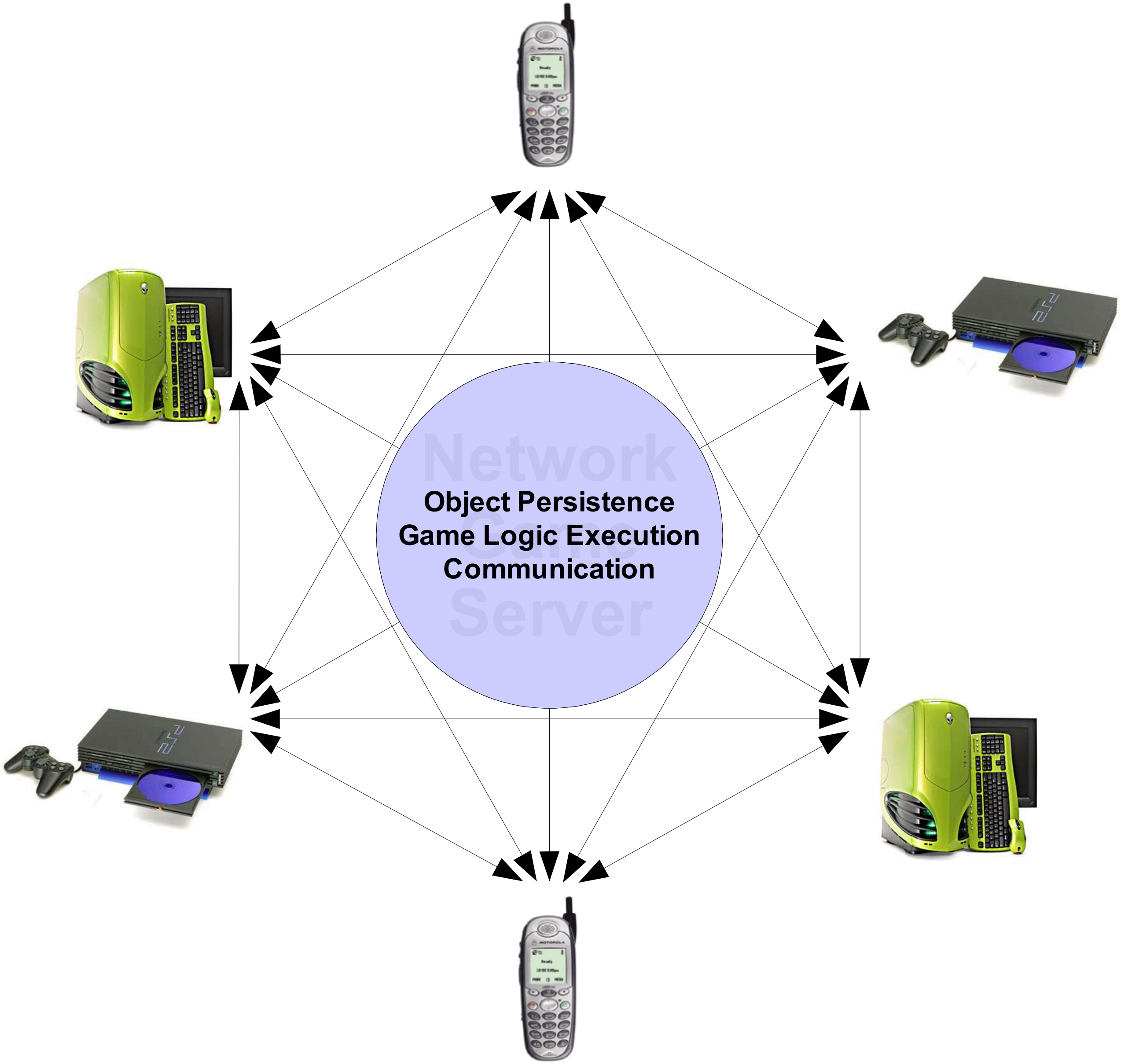
From Game Server to Game Service

java.sun.com/javaone/sf

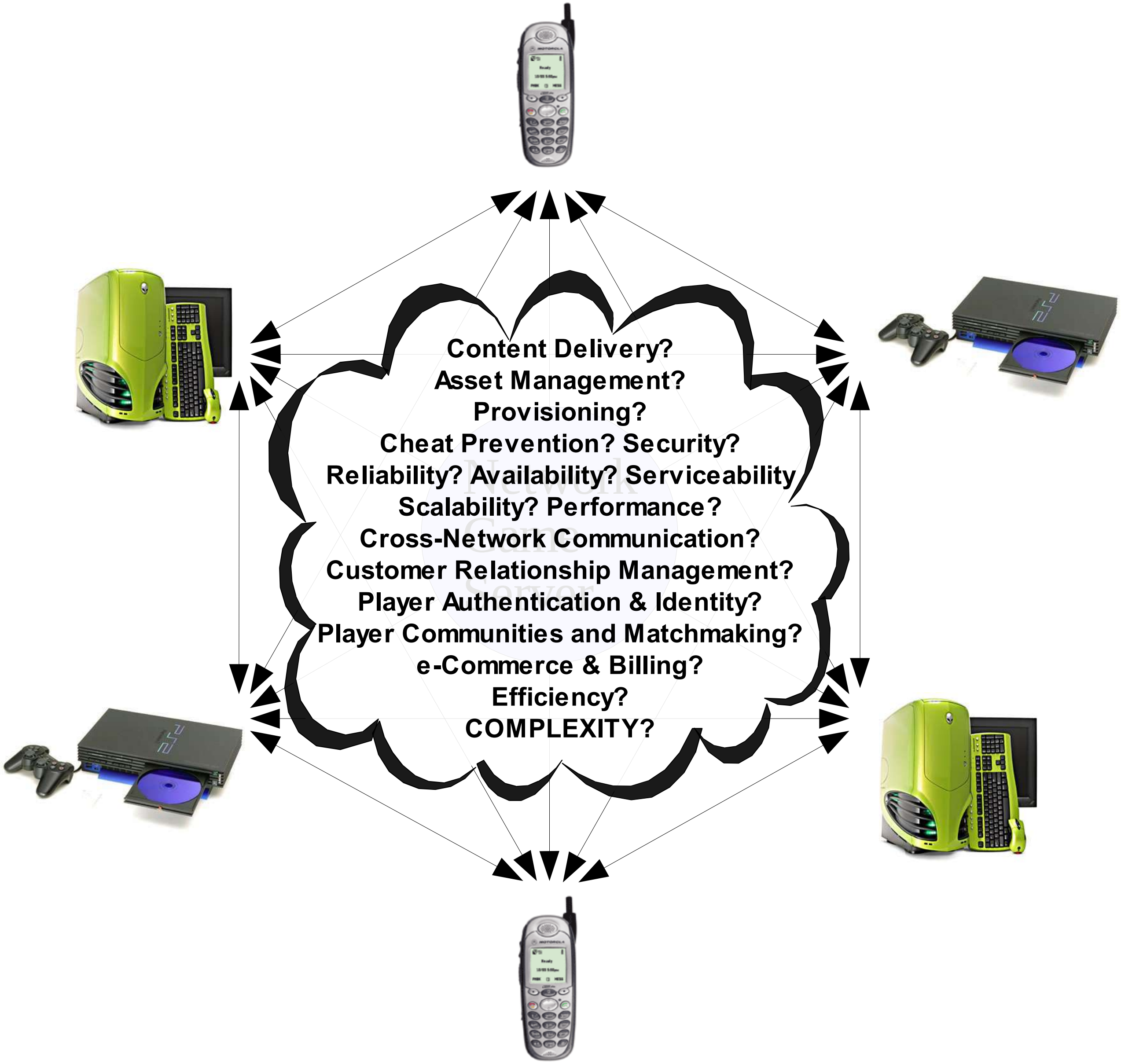
Java™

Sun microsystems

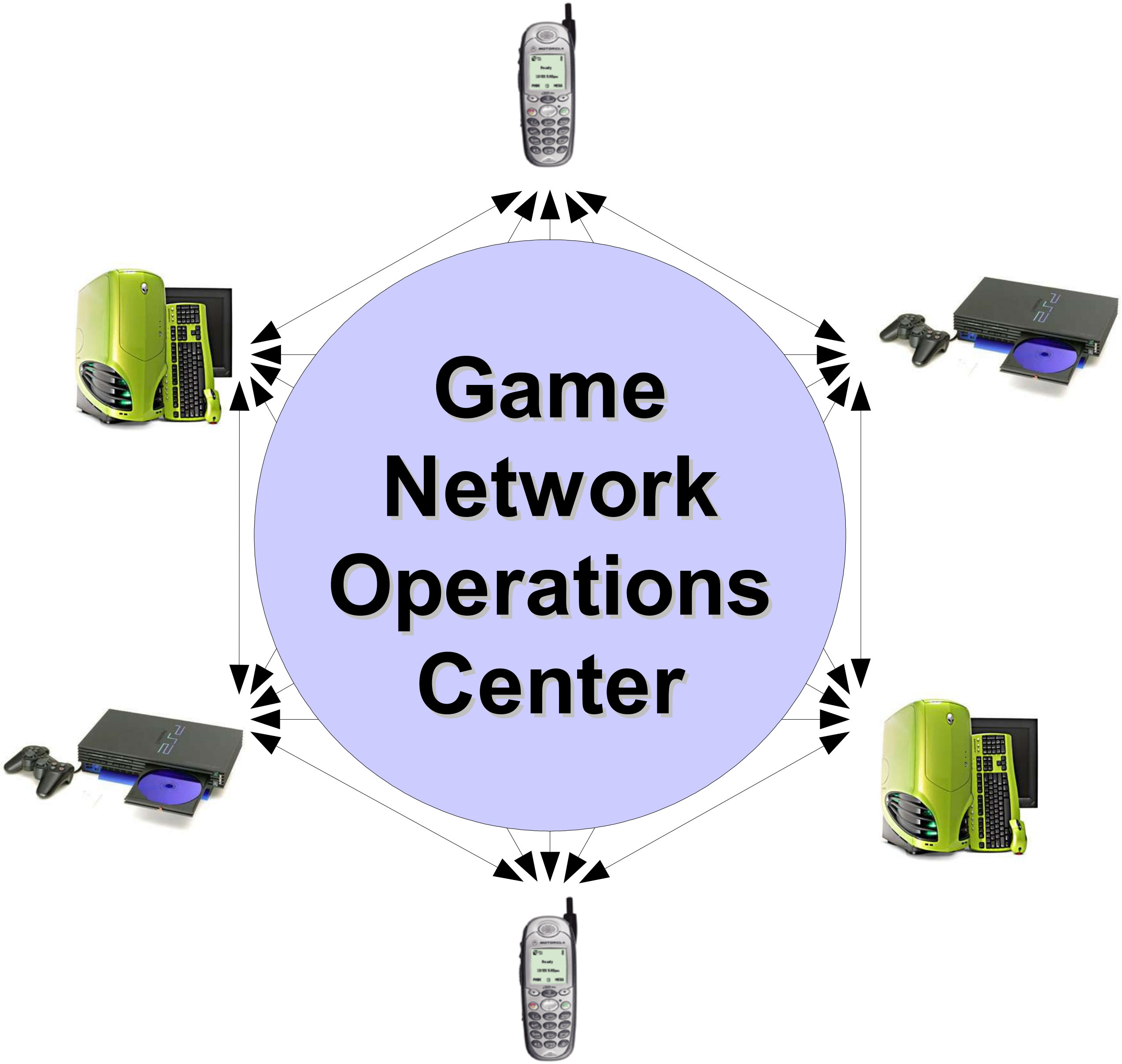

Network
Game
Server



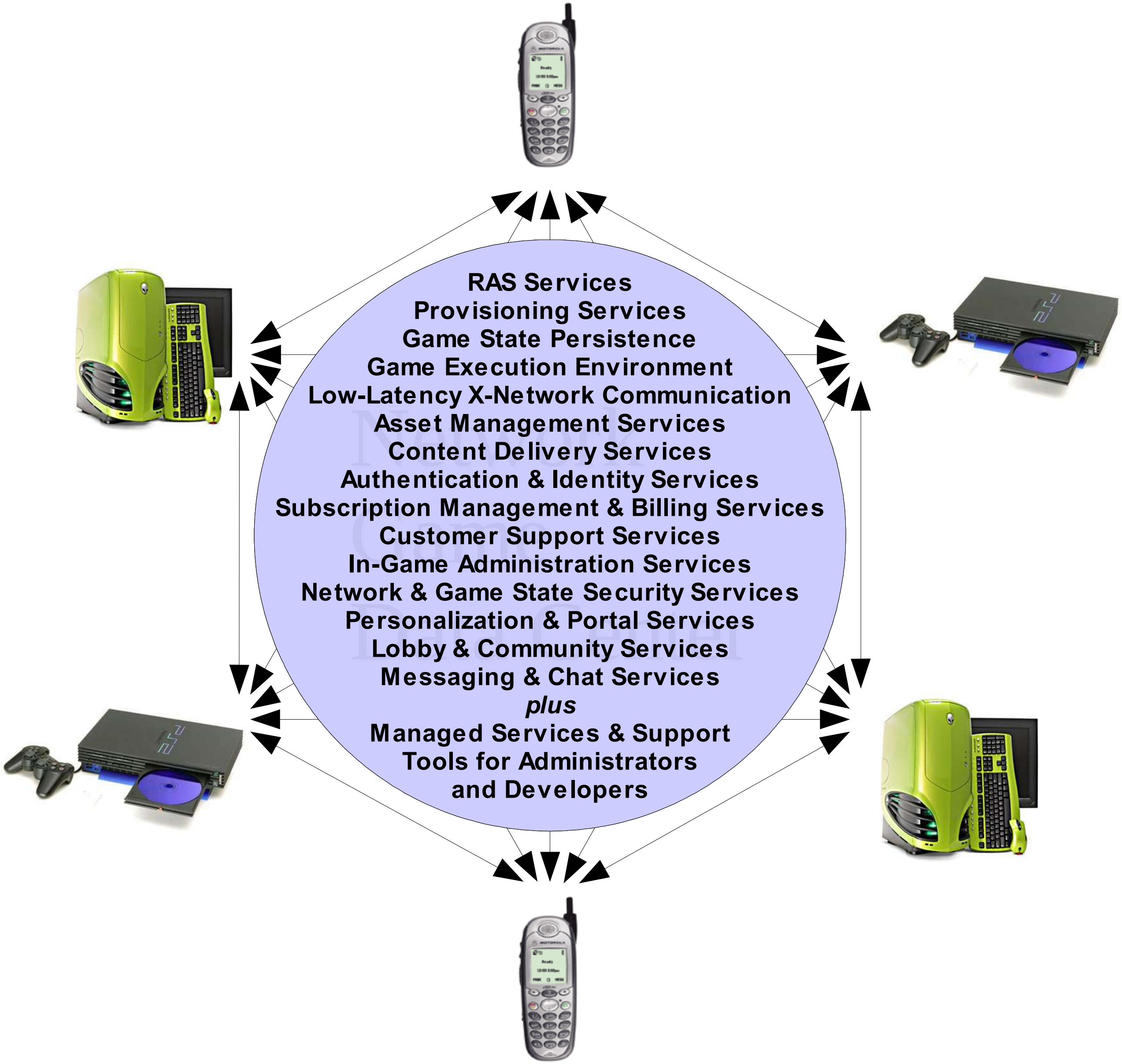
Object Persistence
Game Logic Execution
Communication



Content Delivery?
Asset Management?
Provisioning?
Cheat Prevention? Security?
Reliability? Availability? Serviceability
Scalability? Performance?
Cross-Network Communication?
Customer Relationship Management?
Player Authentication & Identity?
Player Communities and Matchmaking?
e-Commerce & Billing?
Efficiency?
COMPLEXITY?



Game Network Operations Center



RAS Services
Provisioning Services
Game State Persistence
Game Execution Environment
Low-Latency X-Network Communication
Asset Management Services
Content Delivery Services
Authentication & Identity Services
Subscription Management & Billing Services
Customer Support Services
In-Game Administration Services
Network & Game State Security Services
Personalization & Portal Services
Lobby & Community Services
Messaging & Chat Services
plus
Managed Services & Support
Tools for Administrators
and Developers

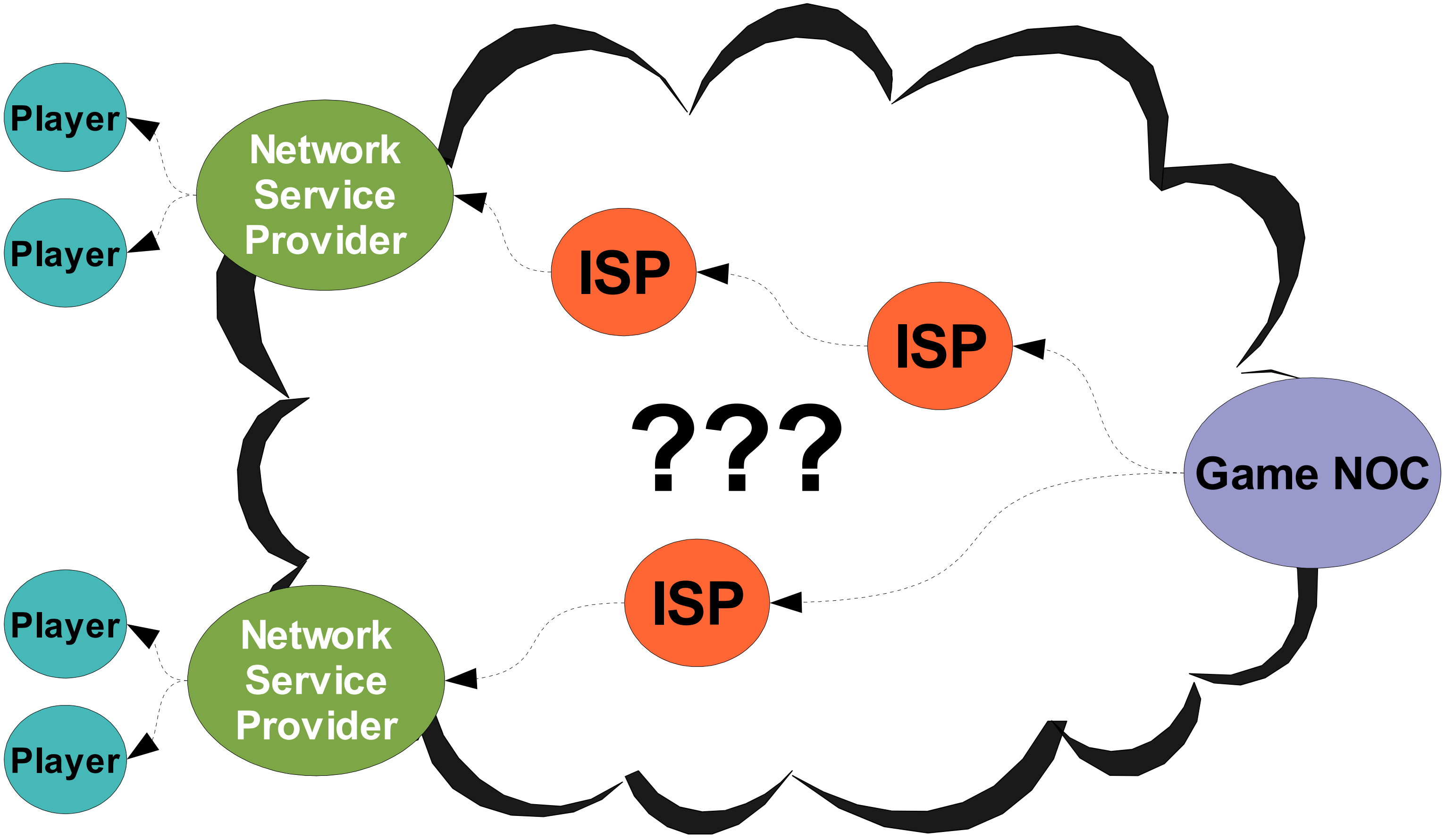
JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

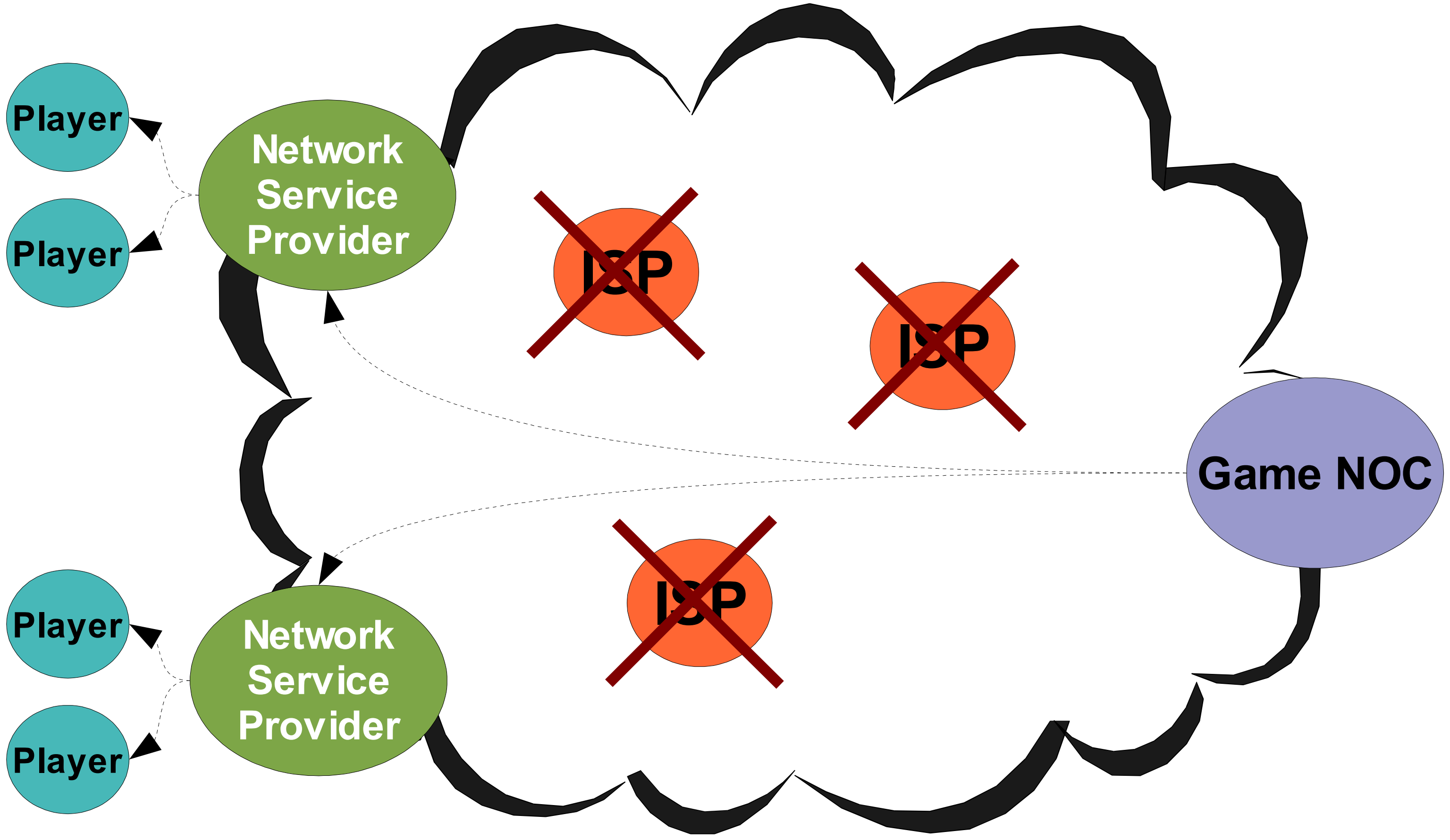
Crossing The Middle Mile

java.sun.com/javaone/sf

Crossing the Middle Mile



Crossing the Middle Mile



JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Opportunity GNOCs

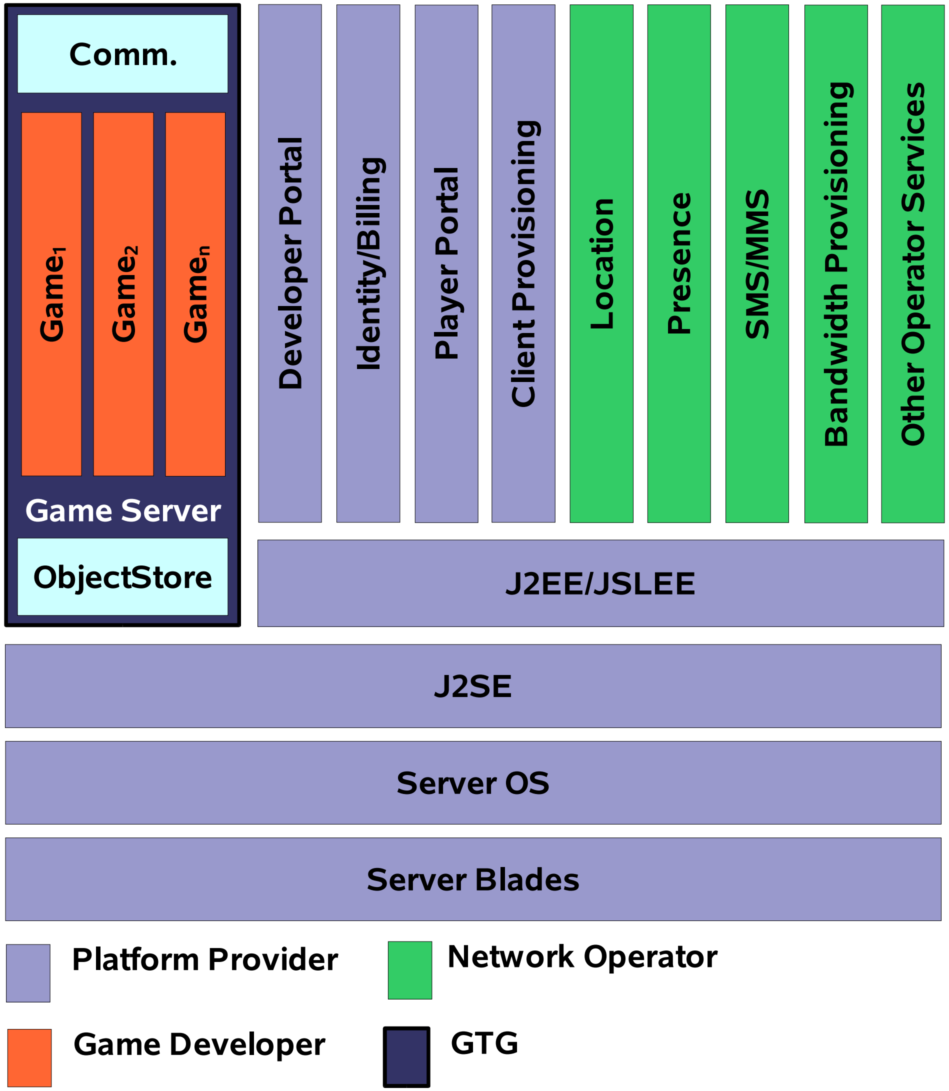
java.sun.com/javaone/sf

Opportunity GNOCs

- MMOGs are EXPENSIVE
  - Getting to launch on an MMOG can take 2 to 3 years and cost over $10M
  - Launch costs (the first 3 months) can run from $3M+ for a moderately successful game
  - Ongoing support costs for network games consume anywhere from 40 to 80 percent of revenues
  - Achieving acceptable performance requires direct-peering relationship with tier 1 providers
- Big opportunity for game hosting services to step in and ease the pain
  - Economies of scale and expertise
  - Requires standard infrastructure to be economical

Opportunity GNOCs

- Scalable Java technology-based game services architecture
  - Capable of supporting multiple games
- Integration with community portals, back-office functions and network operator services



Q&A

Jeff Kesselman
Athomas Goldberg
Sun Game Technology Group

JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

java.sun.com/javaone/sf

Server Architectures for Massively Multiplayer Online Games

Jeffrey Kesselman
Game Server Architect
Game Technology Group
Sun Microsystems, Inc.

JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Additional Slides

java.sun.com/javaone/sf

Launching an MMOG Service

- 2 to 3 years of development
- Getting to launch can run in the seven figures
- Infrastructure costs for launch (first 3 months) can run $2-3M+
- Very little re-use between properties

Sustaining a MMOG Service

Persistant World Support Costs		
Customer Service/Player Relations	**Employees**	**% of Revenues**
24/7 In-game customer representatives	40+	7.00%
Game Masters (storyline, quests, volunteers, etc)	5+	0.75%
Empowered Player Support Team (new player helpers, GM helpers)	5+	0.75%
Other (anti-cheat investigation team, administrative assistant)	3+	0.50%
Community Relations (we site, message boards, email)	5+	0.75%
Game Operations		
Live Development Team (new lands, bug fixes, new features, add-on SKUs)	12-15	6.00%
Network Operations		
NOC Staff	8-12	4.25%
Co-located Server Hardware and Bandwidth		20.00%
Total Maximum Percentage of Revenue Goals		**40.00%**

Source: DFC Intelligence, Online Games Report

Crossing the Middle Mile

- Last mile currently beyond your control
  - Broadband penetration increasing, but not there yet
- Middle Mile becomes focus of network optimization
  - Reduce or eliminate role of ISPs in delivery chain
  - Co-location with Tier-1 Providers becomes essential
- Opportunity for game hosting services
  - Vastly reduce support costs associated with managing online games
  - Requires common infrastructure to be economical

From Game Servers to Game Service

- Not delivering software, but a 24 x 7 x 365 network service
  - Major Infrastructure Investment
  - “Middle-Mile” problem
  - Ongoing support costs consume 40 to 60+ percent of revenue