MMO Server Design with Twisted Python

Dan Maas
SpinPunch Games

dmaas@spinpunch.com

March 16, 2015

Abstract

Our Python server drives top MMO strategy games like *Thunder Run: War of Clans*, with 4 million accounts and over 1,000 concurrent logins. This talk will explain the basic architecture of our server-side software, with special focus on how we use the open-source Twisted library to write high-performance, low-latency network code.

I. Introduction to SpinPunch and our games

- We are a 3-year-old company with a very small engineering team
- One engine drives five game titles on two social platforms
- Top game: *Thunder Run* with 4 million accounts and over 1,000 concurrent players
- Newest game: *Summoner's Gate*, just released on Facebook

II. Requirements for our MMO game engine

- We create "builder" games with deep strategic upgrade paths
- Client-side tactical combat engine
- Game title = Engine + Game Data (JSON) + Art Assets
- Games are closest to a client/server CRUD app with complex business logic, "MMO lite"
- **Scale:** Support 50,000+ daily players, 2,000+ concurrent players
- **Low latency** extremely important (∼200ms hard limit to request latency)
- **High availability**
  - Short-duration unplanned downtime is worse than long-duration planned downtime
  - Very frequent (daily+) deployments of new code and data, requires seamless upgrades
- **Easy** installation, testing, and portability
  - Minimize dependencies
  - Deploy to any well-behaved Linux or Mac OSX system

III. Architecture overview

- "Backing store" for all persistent state in Amazon S3

  - Easy to reason about where state belongs
  - Simple disaster recovery
- Small amount of cross-player information in MongoDB
  - Clans
  - Top Scores
  - Player Info cache
  - Mutex locking
  - Server status
  - Originally was custom Python/Twisted/AMP server, but replaced with MongoDB when we realized it had nearly the same API.
- Lightweight front-end load balancer
  - **Python/Twisted asynchronous HTTP server**
  - Handles platform login process, assigns game server, then gets out of the way
  - Also handles some non-gameplay HTTP API endpoints
    - Facebook Payments, real-time updates, server management, etc
  - Many concurrent "business logic" game server instances
    - **Python/Twisted asynchronous HTTP/WebSockets servers**
    - Up to ~100 concurrent players per CPU core
      - Main bottleneck is JSON parsing/unparsing for login/logout
      - As we add more MMO features, $O(n^2)$ communications patterns will become a problem
  - Lightweight RPC server for cross-game-server messaging
    - **Python/Twisted AMP protocol server**
    - Chat messages
    - Regional Map updates
  - Lightweight web apps for server management, customer support
    - Python CGI scripts
  - Other components (beyond the scope of this talk):
    - The client (JavaScript/HTML5 Canvas)
    - Massive analytics system
    - Display advertising control system

IV. Game servers

- Client speaks directly to these servers to run game logic
  - e.g. "Upgrade this building, produce this unit, buy this thing in the Store"
- Most requests handled synchronously
  - Check requirements, mutate in-memory player state and/or make quick synchronous MongoDB query, then return response
- Anything "slow" (> 100ms) must be asynchronous
  - Reading/writing Amazon S3 files on login/logout
  - Querying Top Scores

V. **Writing an asynchronous HTTP server with Twisted**

- Start with standard twisted.web synchronous request/response model
- `Code sample goes here`
- Return `twisted.web.NOT_DONE_YET`
  - Now you need to call `request.write()` and `request.finish()`
- `Code sample goes here`
- Use Deferred and continuation callbacks to chain code path
- `Code sample goes here`
- Or use new inlineCallbacks generator system
- `Code sample goes here`
- **Monitor in-flight requests**
  - Asynchronous frameworks are bad at this
  - Handle failure/cancel paths, mutex issues
- Many architecture decisions deal with synchronous vs. asynchronous code; what code needs to know/mutate what data when
- Asynchronous code is hard to write and reason about!

VI. **Latency Profiling**

- Key performance metric is not CPU usage, but request latency
- Decorate every entry point with time measurements
- Example of instrumenting HTTP request handler
- Example of instrumenting MongoDB database query
- Collect data on each request
  - Average latency (performance hotspot)
  - Maximum latency (latency hotspot)
- Also monitor total "unhalted" time to judge server load
  - Approaching 50% implies danger of livelock

VII. **Adding WebSockets support**

- Community patch to Twisted
- WebSocket messages call same handlers as HTTP requests
  - Hack: create fake HTTP request for each WebSocket message
- Improved robustness vs. HTTP, but performance did not change
  - HTTP Keepalive is doing its job
- Beware protocol bugs

VIII. **Optional topic: Cross-server RPC with AMP**

- Need to broadcast chat messages and map updates across servers
- HTTP would work, but not ideal

- High overhead per request
- Want to launch some requests synchronously

- AMP advantages:
  - Ships with Twisted
  - Very lightweight
  - Easy to call synchronously and asynchronously
    - Synchronous API via third-party ampy module
- Disadvantages:
  - Not as universal, well-specific, or performant as other systems like Google Protocol Buffers or Thrift

IX. **Optional topic: A domain-specific language for builder games**