Scalable Network Services

Robert Grimm
New York University
Altogether Now: The Three Questions

- What is the problem?
- What is new or different or notable?
- What are the contributions and limitations?
Implementing Global Services: Clusters to the Rescue?

- Advantages
  - Scalability
    - Internet workloads tend to be highly parallel
    - Clusters can be more powerful than biggest iron
    - Clusters can also be expanded incrementally
  - Availability
    - Clusters have “natural” redundancy due to independence of nodes
      - There are no central points of failure in clusters anymore (?)
    - Clusters can be upgraded piecemeal (hot upgrade)
  - Commodity building blocks
    - Clusters of PCs have better cost/performance than big iron
    - PCs are also much easier to upgrade and order
Challenges

- Administration
  - Maintaining lots of PCs vs. one big box can be daunting

- Component vs. system replication
  - To achieve incremental scalability, we need component-level instead of system-level replication

- Partial failures
  - Clusters need to handle partial failures while big iron does not
    - Do we believe this?

- Shared state
  - Managing state across the cluster is harder than for big iron, which can rely on distributed shared memory
BASE Semantics

- Trading between consistency, performance, availability
  - One extreme: ACID
    - Perfect consistency, higher overheads, limited availability
  - An alternative point: BASE
    - Basically available
      - Can tolerate stale data
    - Soft state to improve performance
    - Eventual consistency
      - Can provide approximate answers

- The reality: Different components with different semantics
  - Focus on services with mostly BASE data
Overview of Architecture

- Front ends accept requests and include service logic
- Workers perform the actual operations
- Caches store content before and after processing
- User profile DB stores user preferences (ACID!)
- Manager performs load balancing
- Graphical monitor supports administrators
Software Structured in Three Layers

- **SNS (Scalable Network Service)**
  - To be reused across clusters

- **TACC (Transformation, Aggregation, Caching, Customization)**
  - Easily extensible through building blocks

- **Service**
  - Application-specific
The SNS Support Layer

- Components replicated for availability and scalability
  - Instances are independent from each other, launched on demand
    - Exceptions: Network, resource manager, user profile database
  - Control decisions isolated in front ends
    - Workers are simple and stateless (profile data part of input)
- Load balancing enabled by central manager
  - Load data collected from workers and distributed to FEs
  - Centralization limits complexity
- Overflow pool helps handle bursts
  - User workstations within an organization (is this realistic?)
Timeouts detect failures, peers restart services
- Cached state used to initialize service
- Soft state rebuilt over time

Stubs hide complexity from front ends and workers
- Manager stubs select workers
  - Including load balancing decisions based on hints
- Worker stubs isolate workers from complexities of system
  - Including hiding multi-threaded operation
  - Also report back load information
The TACC Programming Model

- **Transformation**
  - Changes content of a single resource

- **Aggregation**
  - Collects data from several sources

- **Caching**
  - Avoids lengthy accesses to origin servers
  - Avoids repeatedly performing transformations/aggregations

- **Customization**
  - Based on key-value pairs automatically provided w/request

- **Key goal:** Compose workers just like Unix pipes
TranSend Implementation

- Web distillation proxy for Berkeley’s modem pool
- Front ends process HTTP requests
  - Look up user preferences
  - Build pipeline of distillers
- Centralized manager tracks distillers
  - Announces itself on IP multicast group
  - Collects load information as running estimate
  - Performs lottery-scheduling between instances
    - Actually, the scheduling is performed by manager stub
  - Spawns new instances of distillers
    - May utilize overflow pool during bursts
Fault tolerance and recovery provided by peers

- Based on broken connections, timeouts, beacons
  - For former, distillers register with manager
- Manager (re)starts distillers and front ends
- Front ends restart manager

Overall scalability and availability aided by BASE

- Load balancing data is (slightly) stale
  - Timeouts allow recovery from incorrect choices
- Content is soft state and can be recovered from cache/server
- Answers may be approximate (on system overload)
  - Use cached data instead of recomputing data
## Comparison between TranSend and HotBot

<table>
<thead>
<tr>
<th>Component</th>
<th>TranSend</th>
<th>HotBot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load balancing</td>
<td>Dynamic, by queue lengths at worker nodes</td>
<td>Static partitioning of read-only data</td>
</tr>
<tr>
<td>Application layer</td>
<td>Composable TACC workers</td>
<td>Fixed search service application</td>
</tr>
<tr>
<td>Service layer</td>
<td>Worker dispatch logic, HTML / JavaScript UI</td>
<td>Dynamic HTML generation, HTML UI</td>
</tr>
<tr>
<td>Failure management</td>
<td>Centralized but fault-tolerant using process-peers</td>
<td>Distributed to each node</td>
</tr>
<tr>
<td>Worker placement</td>
<td>FE’s and caches bound to their nodes</td>
<td>All workers bound to their nodes</td>
</tr>
<tr>
<td>User profile (ACID) database</td>
<td>Berkeley DB with read caches</td>
<td>Parallel Informix server</td>
</tr>
<tr>
<td>Caching</td>
<td>Harvest caches store pre- and post-transformation Web data</td>
<td>Integrated cache of recent searches, for incremental delivery</td>
</tr>
</tbody>
</table>
TranSend Evaluation

- Based on traces from modem pool
  - 25,000 users
  - 600 modems (14.4K and 28.8K)
  - 20 million HTTP requests (over 1 1/2 months)

- Distribution of content sizes
  - Most content accessed is small
  - But average byte part of large content
  - Therefore, distillation makes sense

- Evaluation of TranSend based on trace playback engine, isolated cluster
Burstiness of Requests

- Three different time scales
  - 24 hours
  - 3.5 hours
  - 3.5 minutes

- How to manage overflow pool?
  - Select utilization level for worker pool
  - Or, select percent time for overflow pool

- Are these the right strategies?
Not surprisingly, grows with input size: 8ms/1KB
- However, with large variation
Cache Partition Performance

**Summary of results**
- Average cache hit takes 27ms, 15ms of which used for TCP
  - What’s wrong with this picture?
- 95% of all cache hits take less than 100ms to service
- Miss penalty varies widely: 100ms to 100s

**Some observations**
- Cache hit rate grows with cache size, but plateaus for population size (why?)
- Cache hit rate also grows with population size, even for fixed-size cache (why?)
- What do we conclude for front end capacity?
Self-Tuning and Load Balancing

- Spawning threshold $H$, stability period $D$
What is the trade-off for D?
### Scalability under Load

- **What are the most important bottlenecks?**
- **How can we overcome them?**

<table>
<thead>
<tr>
<th>Requests/Second</th>
<th># Front Ends</th>
<th># Distillers</th>
<th>Element that saturated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-24</td>
<td>1</td>
<td>1</td>
<td>distillers</td>
</tr>
<tr>
<td>25-47</td>
<td>1</td>
<td>2</td>
<td>distillers</td>
</tr>
<tr>
<td>48-72</td>
<td>1</td>
<td>3</td>
<td>distillers</td>
</tr>
<tr>
<td>73-87</td>
<td>1</td>
<td>4</td>
<td>FE Ethernet</td>
</tr>
<tr>
<td>88-91</td>
<td>2</td>
<td>4</td>
<td>distillers</td>
</tr>
<tr>
<td>92-112</td>
<td>2</td>
<td>5</td>
<td>distillers</td>
</tr>
<tr>
<td>113-135</td>
<td>2</td>
<td>6</td>
<td>distillers &amp; FE Ethernet</td>
</tr>
<tr>
<td>136-159</td>
<td>3</td>
<td>7</td>
<td>distillers</td>
</tr>
</tbody>
</table>
What Do You Think?