Programming in the Cloud

Babak Behzad
Faraz Faghri
... Big Data

Everybody fancies:
- data analysis
- log processing
- machine learning
- spam detection
...

But ...
- unstructured data
- complex infrastructure
- MapReduce coding is time-consuming
- and long
...

Data analyst asks ...

- what are the top 5 most visited pages by users aged 18-25?
MapReduce solution ...
Desired solution ...

- Desirable to have a higher level declarative language that describes the parallel data processing model.
- SQLish query where the user specifies the "what" and leaves the "how" to the underlying processing engine.
Pig and Hive: 2 ways of doing 1 thing ...

- Both generate Map-Reduce jobs from a query written in a higher level language.
- User doesn't need to deal with Hadoop complexity.
- Lives on client machine, nothing to install on cluster.

users = load 'users';
grouped = group users by zipcode;
byzip = foreach grouped
    generate zipcode, COUNT(users);
store byzip into 'count_by_zip';

Map Reduce Job:
Input: ./users
Map: project(zipcode, userid)
Shuffle key: zipcode
Reduce: count
Output: ./count_by_zip
## Pig and Hive example ...

<table>
<thead>
<tr>
<th></th>
<th>Procedural data-flow language</th>
<th>Declarative SQLish language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Count</strong></td>
<td>- mytable = GROUP mytable ALL;</td>
<td>- SELECT COUNT(*) FROM mytable;</td>
</tr>
<tr>
<td></td>
<td>- mytable = FOREACH mytable GENERATE COUNT (mytable);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- DUMP mytable;</td>
<td></td>
</tr>
<tr>
<td><strong>Count Distinct Number</strong></td>
<td>- mytable_col1 = GROUP mytable BY col1;</td>
<td>- SELECT COUNT(DISTINCT col1) FROM mytable;</td>
</tr>
<tr>
<td></td>
<td>- mytable_col1_count= FOREACH mytable_col1 {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>col_2 = DISTINCT mytable.col_2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GENERATE group, COUNT(col_2);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>};</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- DUMP mytable_col1_count;</td>
<td></td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>- mytable = JOIN mytable BY col1, othertable BY col1;</td>
<td>- SELECT * FROM mytable INNER JOIN othertable ON mytable.col1 = othertable.col1;</td>
</tr>
<tr>
<td></td>
<td>- DUMP mytable;</td>
<td></td>
</tr>
</tbody>
</table>
Let's analyze tweets ...

- **Twitter Streaming API** outputs **tweets** in a **JSON format**
- **Flume**: distributed service for efficiently collecting, aggregating, and moving large amounts of log data.
- **Hive** provides a query interface which can be used to query data that resides in HDFS.
- **SerDe** stands for Serializer and Deserializer, which are interfaces that tell Hive how it should translate the data into something that Hive can process.
- Apache Oozie is a workflow coordination system, an extremely flexible system for designing **job workflows**, which can be **scheduled to run** based on a set of criteria.
Pig ...

pig.jar:
- parses
- checks
- optimizes
- plans execution
- submits jar to Hadoop
- monitors job progress

Pig Latin

```pig
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A by y > 0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE
    group, COUNT(D);
STORE F INTO 'output';
```

Logical Plan

```
LOAD (x, y, z)
FILTER (x, y, z)
JOIN (x, y, z, t, u, v)
GROUP (group, ((x, y, z, t, u, v)))
FOREACH (group, count)
STORE (group, count)
```
Hive ...

- **Tables**: Each table has a corresponding directory on HDFS.
- **Partitions**: distribution of data within sub-directories of a table directory.
- **Buckets**: files in partition dir, based on the hash of a column in the table.
Hive ...

- Apache **Thrift** software framework, for scalable cross-language services development
- **DDL**: Hive Data Definition Language: create table, show, alter ...
User Defined Function ...

- way to do an operation on a field or fields
- called from within a pig script
- \( b = \text{FOREACH} \ a \ \text{GENERATE} \ \text{foo}(\text{color}) \)

- Implementation: just think about a small problem
- Many collaborators: Linkedin DataFU, Twitter Elephant-bird, etc.
Performance ...

**Twitter**: Typically a Pig script is 5% of the code of native MapReduce written in about 5% of the time. However, queries typically take between 110-150% the time to execute that a native MapReduce job would have taken.

**Hortonworks benchmark:**
Final word ...

- Not a database, read-only operations.
- Load data you tell how the data is organized, deserialize the data.
- They work with Hadoop and HDFS.

Piazza
- Reliability
- Popularity
MegaPipe: A New Programming Interface for Scalable Network I/O

OSDI'12
Imagine a typical Web 2.0 data-center (e.g. facebook, Twitter, Wikipedia, Youtube, etc.)

- web applications access the database to retrieve and update user data

- However, the database can become a bottleneck.

- ... so a cache is used to store the results of database queries.
- cache itself must be highly scalable

- Memcached is an example of a scale-out system: adding additional servers to a Memcached cluster linearly increases the capacity of the cache.
Memcached:
- provides a giant distributed hash table.
- When the table is full: LRU order.
- Applications
  - first check memcached (RAM)
  - then fetch from slower backing store (DB, Disk)

```php
function get_foo(int userid) {
    data = db_select("SELECT * FROM users WHERE userid = ?", userid);
    return data;
}
```

```php
function get_foo(int userid) {
    /* first try the cache */
    data = memcached_fetch("userrow:" + userid);
    if (!data) {
        /* not found : request database */
        data = db_select("SELECT * FROM users WHERE userid = ?", userid);
        /* then store in cache until next get */
        memcached_add("userrow:" + userid, data);
    }
    return data;
}
```
Early Motivation

If you've read anything about scaling large websites, you've probably heard about memcached. memcached is a high-performance, distributed memory object caching system. Here at Facebook, we're likely the world's largest user of memcached. We use memcached to alleviate database load. memcached is already fast, but we need it to be faster and more efficient than most installations. We use more than 800 servers supplying over 28 terabytes

1. Per-connection buffer to read, write data: a lot of memory => a per thread shared connection buffer pool
2. Considerable lock contention over UDP socket locks => one reply socket per thread
3. One core get saturated, doing network interrupt handling => distribute network processing evenly across cores.
4. Their machines: 8-core. A big contention on the lock that protects each network device’s transmit queue => better algorithm with less locking
Motivation

1. Bulk-transfer workload
   - One way, large data transfer
     - Video streaming, HDFS
   - Cheap

2. Message-oriented workload
   - Short connections or small messages
     - HTTP, RPC, DB, key-value stores, ...
   - CPU-intensive

Existing network API on multi-core systems
   - Inefficient for "message-oriented" workloads
   - Hard time to scale to high connection rate
Low throughput of small messages

Not efficient use of multi-cores
• Networking: Central role in modern applications

• MegaPipe: A new API for efficient, scalable network I/O. Especially I/O intensive workloads

• More on Operating System, not Distributed Systems

• The core abstraction: a channel
  ○ a per-core, bi-directional pipe between the kernel and
The image contains a diagram with nodes labeled as:

- **Lightweight socket**
- **Listening socket partitioning**
- **File abstraction overhead (VFS overhead)**
- **Shared listening socket**

The diagram also includes a code snippet:

```c
n_events = epoll_wait(...);
for (...) {
    ...
    new_fd = accept(listen_fd);
    ...
    bytes = recv(fd2, buf, 4096);
    ...
}
```

Commentary:

- `for (...) {` // wait for I/O readiness
- `new_fd = accept(listen_fd);` // new connections
- `bytes = recv(fd2, buf, 4096);` // new data for fd2

The diagram illustrates the relationships between system call overheads and the operations involved in handling I/O events and new connections.
Architectural Overview

- MegaPipe involves both user-space library and Linux kernel modifications.
Listening Socket Partitioning

- When an application thread associates a listening socket to a channel, MegaPipe spawns a separate listening socket.
lwsocket: Lightweight Socket

- In UNIX, many different types of open files:
  - disk files, sockets, pipes, devices
- This is an overkill for network sockets:
  - Sockets are rarely shared
  - Sockets are ephemeral
- Lwsocket: Do not create a file instance
System Call Batching

- **System calls:**
  - Cost of mode switching => Expensive
  - Negative effect on cache locality => Inefficient

- To amortize system call costs, MegaPipe:
  - Batches multiple I/O requests and their completion notifications into a single system call.
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp_create()</td>
<td></td>
<td>Create a new MegaPipe channel instance.</td>
</tr>
<tr>
<td>mp_register()</td>
<td>channel, fd, cookie, cpu_mask</td>
<td>Create a MegaPipe handle for the specified file descriptor (either regular or lightweight) in the given channel. If a given file descriptor is a listening socket, an optional CPU mask parameter can be used to designate the set of CPU cores which will respond to incoming connections for that handle.</td>
</tr>
<tr>
<td>mp_unregister()</td>
<td>handle</td>
<td>Remove the target handle from the channel. All pending completion notifications for the handle are canceled.</td>
</tr>
<tr>
<td>mp_accept()</td>
<td>handle, count, is_lwsocket</td>
<td>Accept one or more new connections from a given listening handle asynchronously. The application specifies whether to accept a connection as a regular socket or a lwsocket. The completion event will report a new FD/lwsocket and the number of pending connections in the accept queue.</td>
</tr>
<tr>
<td>mp_read()</td>
<td>handle, buf, size</td>
<td>Issue an asynchronous I/O request. The completion event will report the number of bytes actually read/written.</td>
</tr>
<tr>
<td>mp_write()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mp_disconnect()</td>
<td>handle</td>
<td>Close a connection in a similar way with shutdown(). It does not deallocate or unregister the handle.</td>
</tr>
<tr>
<td>mp_dispatch()</td>
<td>channel, timeout</td>
<td>Retrieve a single completion notification for the given channel. If there is no pending notification event, the call blocks until the specified timer expires.</td>
</tr>
</tbody>
</table>

Table 1: MegaPipe API functions (not exhaustive).
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=0x01)
mp_accept(handle)

while True:
ev = mp_dispatch(ch)
conn = ev.cookie
if ev.cmd == ACCEPT:
    mp_accept(conn.handle)
    conn = new Connection()
    conn.handle = mp_register(ch, ev.fd,
                              cookie=conn)
    mp_read(conn.handle, conn.buf, READSIZE)
elif ev.cmd == READ:
    mp_write(conn.handle, conn.buf, ev.size)
elif ev.cmd == WRITE:
    mp_read(conn.handle, conn.buf, READSIZE)
elif ev.cmd == DISCONNECT:
    mp_unregister(ch, conn.handle)
delete conn

Listing 1: Pseudocode for ping-pong server event loop
Figure 4: Relative performance improvement for varying message sizes over a range of CPU cores.
Microbenchmarks: Multi-core scalability

**Figure 3:** Comparison of parallel speedup for varying numbers of transactions per connection (labeled) over a range of CPU cores (x-axis) with 64 B messages.
Figure 5: memcached throughput comparison with eight cores, by varying the number of requests per connection. $\infty$ indicates persistent connections. Lines with "X" markers (-FL) represent fine-grained-lock-only versions.
MacroBenchmarks: nginx

Figure 7: Evaluation of nginx throughput for the (a) SpecWeb, (b) Yahoo, and (c) Yahoo/2 workloads.
Conclusion

● Short connections or small messages:
  ○ High CPU overhead
  ○ Poorly scaling with multi-core CPUs

● MegaPipe
  ○ Key abstraction: per-core channel
  ○ Enabling three optimization opportunities:
    ■ Batching, partitioning, lwsocket
Thoughts/criticism

- **Multi-core Era!**
- **Legacy Code:** The main question is, are people willing to change their code to this new API?
- **Evaluation?** Why not more real-world applications?
Discussion Points

● From Piazza:
  ○ Compared to HW solutions (infiniband)
  ○ Delay of batching?
  ○ Will it be as great as it is right now for VMs?
  ○ All the API is asynchronous!
  ○ "applications should control sharing"

● More?
  ○ Let's discuss here in the class...
Implementation

- MegaPipe consists of two parts:
  - Kernel Module: MP-K
  - User-level Library: MP-L
  - Interaction: Through a special device: /dev/megapipe

- MP-K
  - Maintains set of handles for both FDs and lwsockets in a red-black tree for each channel.
  - Depending on the file type, it handles async I/O.

- Upon receiving batched I/O commands from MP-L through a channel, MP-K checks if they can be processed immediately.
  - If yes, process it and issue a completion notification
  - If no, bookkeep, register a callback to the socket or declare epoll interest for other file types.