Scaling massive, real-time data pipelines with Go

Jean de Klerk

github.com/jadekler/git-go-scalability-talk
What kind of pipelines?

- Highly available (HA)
- Very low message drop rate
- High throughput: 20k+ messages per second
Guiding principles

Highly Available
- Stateless
- Horizontally scaled microservices

Low Drop Rate
- Stateless
- Flush capability

High Throughput
- Horizontally scaled microservices
- Highly concurrent
What many data pipelines look like

IOT

Gateway
Ingestion
Processing
Parsing
Passing Along

Gateway

Gateway

Gateway

Ingestion
Processing
Parsing
Passing Along

Services

IOT Services
What many data pipelines look like

Messaging

Gateway
Ingestion
Processing
Parsing
Passing Along

Gateway

Gateway

Gateway

Messaging
What many data pipelines look like

Logs/Metrics

Gateway
Ingestion
Processing
Parsing
Passing Along

Gateway
Ingestion
Processing
Parsing
Passing Along

Gateway
Ingestion
Processing
Parsing
Passing Along

Database

SPARK
MySQL
PostgreSQL
Hadoop
Optimization opportunities

#1: IO

#2: Application

#1: IO
Application Structure

type Queue interface {
    Enqueue(data []byte)
    Dequeue() ([]byte, bool)
}

type inputter interface {
    StartAccepting(q queues.Queue)
}

type outputter interface {
    StartOutputting(q queues.Queue)
}

type Processor struct {
    i inputter
    q queues.Queue
    o outputter
    wg *sync.WaitGroup
}

func (p *Processor) Start() {
    go p.i.StartAccepting(p.q)
    go p.o.StartOutputting(p.q)
    p.wg.Wait()
}
Let’s talk about IO

#1: IO
#2: Application

#1: IO
HTTP API
HTTP API

• 👍 Less lossy, easy to know if connection succeeded

• 👎 Unary connections are slow
# One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC unary</th>
<th>Websocket streaming</th>
<th>gRPC streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td></td>
<td>74,761</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td></td>
<td>90,750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td></td>
<td>106,020</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td></td>
<td>298,588</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
UDP
UDP

- 👍 Since it doesn’t wait for response, fairly fast
- 👎 Lossy
- 👎 Hard to know how much is being lost
- 👎 Unary connections are slow
# One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC Unary</th>
<th>Websocket Streaming</th>
<th>gRPC Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>74,761</td>
<td>71,157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>90,750</td>
<td>77,490</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td>106,020</td>
<td>74,508</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very Large</td>
<td>298,588</td>
<td>N/A *</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* UDP max size = 65,507 bytes, “Very Large” size = 94,943 bytes
HTTP2 unary gRPC
HTTP/2 unary gRPC

• 👍 Protobuf native = binary-encoded data

• 👍 Protobuf native = Service and model language-agnostic spec

• 👍 HTTP/2 is great

• 👍 Built with load balancing in mind

• 👎 Unary connections are slow
## One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC unary</th>
<th>Websocket streaming</th>
<th>gRPC streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>74,761</td>
<td>71,157</td>
<td>110,169</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>90,750</td>
<td>77,490</td>
<td>125,060</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>106,020</td>
<td>74,508</td>
<td>151,361</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>298,588</td>
<td>N/A *</td>
<td>390,081</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* UDP max size = 65,507 bytes, “Very Large” size = 94,943 bytes
TCP Websockets
TCP Websockets

- 👍👍 Streaming is super fast
- 👎👎 Hard to know if connection is alive
- 👎  Load balancing can be tricky
# One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC unary</th>
<th>Websocket streaming</th>
<th>gRPC streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>74,761</td>
<td>71,157</td>
<td>110,169</td>
<td>5,572</td>
<td></td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>90,750</td>
<td>77,490</td>
<td>125,060</td>
<td>14,200</td>
<td></td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>106,020</td>
<td>74,508</td>
<td>151,361</td>
<td>17,794</td>
<td></td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>298,588</td>
<td>N/A *</td>
<td>390,081</td>
<td></td>
<td>189,405</td>
</tr>
</tbody>
</table>

* UDP max size = 65,507 bytes, “Very Large” size = 94,943 bytes
HTTP2 streaming gRPC
HTTP2 streaming gRPC

- 👍👍 Streaming is super fast
- 👍 Protobufs (smaller packets, service/model standard)
- 👍 HTTP/2 is great
- 👍 Built with load balancing in mind
- 👎👎 Hard to know if connection is alive
## One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC unary</th>
<th>Websocket streaming</th>
<th>gRPC streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>74,761</td>
<td>71,157</td>
<td>110,169</td>
<td>5,572</td>
<td>9,017</td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>90,750</td>
<td>77,490</td>
<td>125,060</td>
<td>14,200</td>
<td>19,845</td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>106,020</td>
<td>74,508</td>
<td>151,361</td>
<td>17,794</td>
<td>27,546</td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>298,588</td>
<td>N/A *</td>
<td>390,081</td>
<td>189,405</td>
<td>276,374</td>
</tr>
</tbody>
</table>

* UDP max size = 65,507 bytes, “Very Large” size = 94,943 bytes
## Many-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>HTTP</th>
<th>UDP</th>
<th>gRPC unary</th>
<th>Websocket streaming</th>
<th>gRPC streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>54,565</td>
<td>68,576</td>
<td>40,430</td>
<td>6,456</td>
<td>3,197</td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>100,904</td>
<td>N/A **</td>
<td>50,723</td>
<td>10,917</td>
<td>15,091</td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>102,188</td>
<td>N/A **</td>
<td>57,708</td>
<td>12,469</td>
<td>19,928</td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>220,838</td>
<td>N/A *</td>
<td>238,559</td>
<td>116,447</td>
<td>150,077</td>
</tr>
</tbody>
</table>

* UDP max size = 65,507 bytes, “Very Large” size = 94,943 bytes

** Too lossy to accurately measure
Streaming Data is 👍 *

* Generally
Databases, Datastores, Datalakes, etc.

• Do what works
LOAD DATA INFILE
Let’s talk about the application

#1: IO

#2: Application

#1: IO
Array-backed queue using mutexes
Array-backed queue using mutices

- 👎 Mutex constant lock/unlock
- 👎 Constant array resizing
- 👎 Array eventually reaches maximum size (if your reads are slower than your writes)
## One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>Arrays + Mutices</th>
<th>Channels</th>
<th>Ring Buffer + Mutices</th>
<th>Ring Buffer + Atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>611</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td></td>
<td>3,352</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td></td>
<td></td>
<td>4,791</td>
<td></td>
</tr>
<tr>
<td>Very Large</td>
<td></td>
<td></td>
<td></td>
<td>16,363</td>
</tr>
</tbody>
</table>
Channel-backed queue
Channel-backed queue

- 👍 Super fast
- 👎👎 Channels eventually fill up and wait for reads (if your reads are slower than writes)
## One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>Arrays + Mutices</th>
<th>Channels</th>
<th>Ring Buffer + Mutices</th>
<th>Ring Buffer + Atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>611</td>
<td>365</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>3,352</td>
<td>1,023</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>4,791</td>
<td>1,616</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>16,363</td>
<td>9,759</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ring buffer backed queue using mutices
Ring buffer backed queue using mutices

- 👍 Ring buffer will automatically flush if writes exceed reads
- 👎 Mutex constant lock/unlock
# One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>Arrays + Mutices</th>
<th>Channels</th>
<th>Ring Buffer + Mutices</th>
<th>Ring Buffer + Atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>611</td>
<td>365</td>
<td>611</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>3,352</td>
<td>1,023</td>
<td>3,797</td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td>4,791</td>
<td>1,616</td>
<td>6,113</td>
<td></td>
</tr>
<tr>
<td>Very Large</td>
<td>16,363</td>
<td>9,759</td>
<td>16,392</td>
<td></td>
</tr>
</tbody>
</table>
Ring buffer backed queue using atomics
Ring buffer backed queue using atomics

- 😊 Ring buffer will automatically flush if writes exceed reads
# One-To-One Comparison

<table>
<thead>
<tr>
<th></th>
<th>Arrays + Mutices</th>
<th>Channels</th>
<th>Ring Buffer + Mutices</th>
<th>Ring Buffer + Atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small</strong></td>
<td>611</td>
<td>365</td>
<td>611</td>
<td>227</td>
</tr>
<tr>
<td><strong>Medium</strong></td>
<td>3,352</td>
<td>1,023</td>
<td>3,797</td>
<td>1,091</td>
</tr>
<tr>
<td><strong>Large</strong></td>
<td>4,791</td>
<td>1,616</td>
<td>6,113</td>
<td>1,611</td>
</tr>
<tr>
<td><strong>Very Large</strong></td>
<td>16,363</td>
<td>9,759</td>
<td>16,392</td>
<td>11,382</td>
</tr>
</tbody>
</table>
End result

• Streaming IO

• Atomic ring buffers
Afterthoughts

• Metrics over logs 📊
Afterthoughts

• Metrics over logs 📊

• Queues are 👍
Afterthoughts

• Metrics over logs 📊

• Queues are 👍

• Cycle long-lived connections ☠
Thank you

Jean de Klerk

github.com/jadekler/git-go-scalability-talk