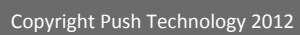




- # About me?

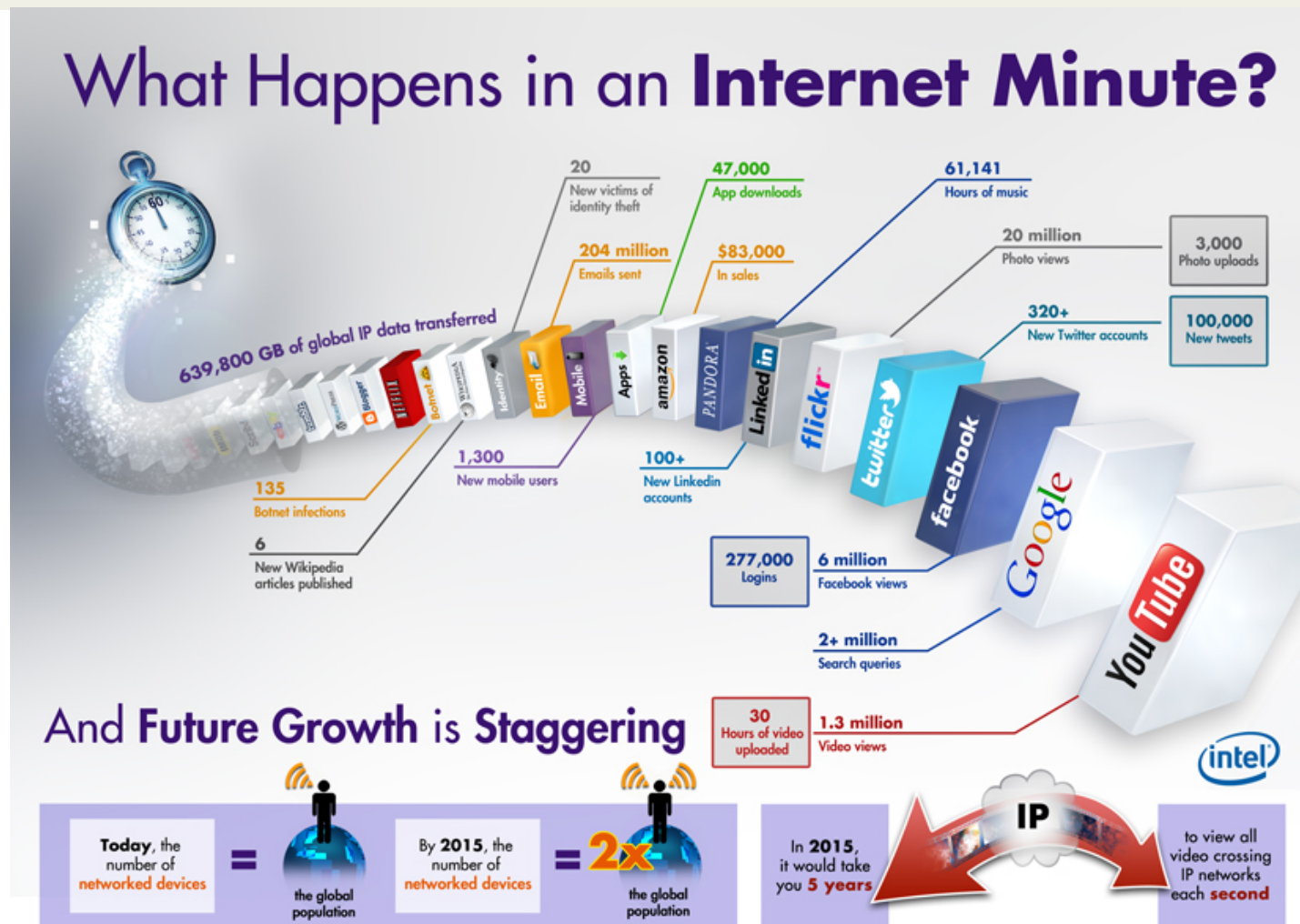


Big Data. The most important V is Value.

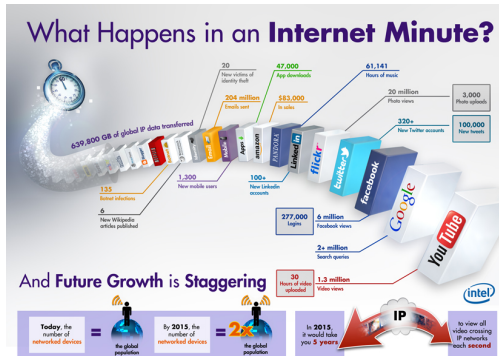
- 4Vs:
 - Volume
 - Velocity
 - Variety
 - Variability
- “It’s the connections that matter most”
- David Evans, Cisco

Number of Internet objects		Number of theoretical connections (0.001%)	
1		-	
100		0	
1,000		5	
1,000,000		4,999,995	
1,000,000,000		4,999,999,995,000	
(Today)	10,000,000,000	499,999,999,950,000	
(By 2020)	51,597,803,520	13,311,666,640,184,600	

Connect all the things



An Internet Minute



10k, 1ms

1k, 100us

Hours
Minutes
Seconds
Millis

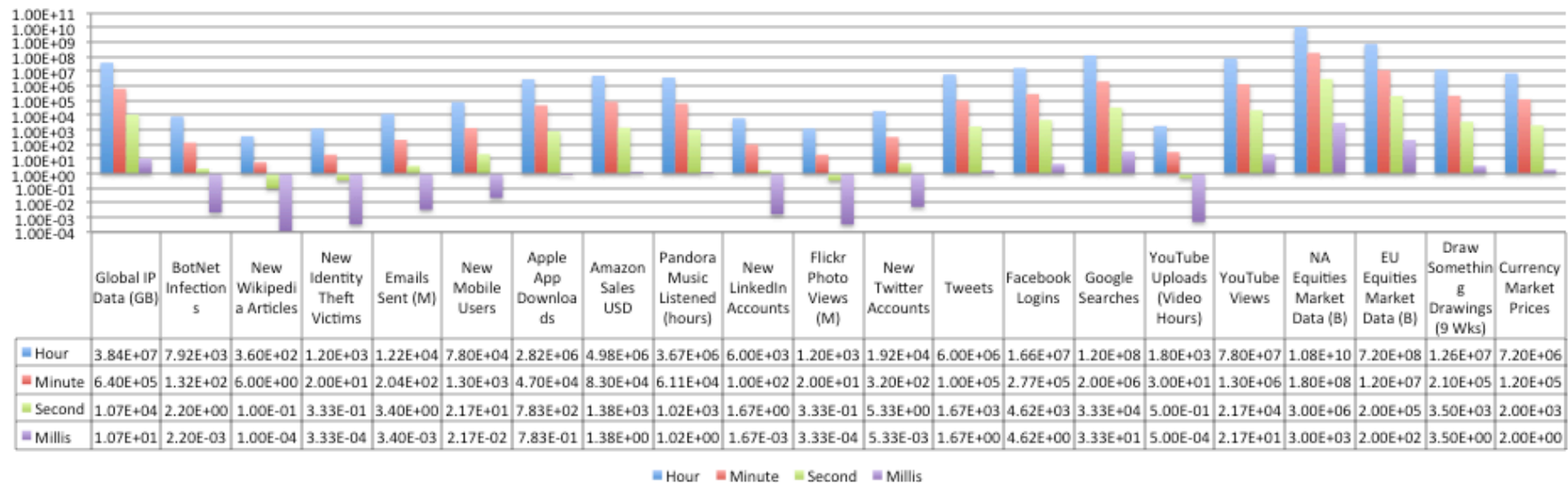


12600000
210000
3500
3.5

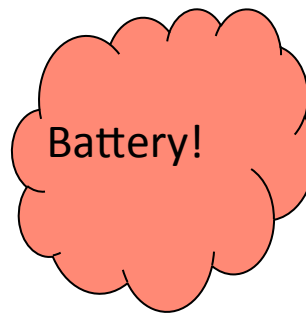
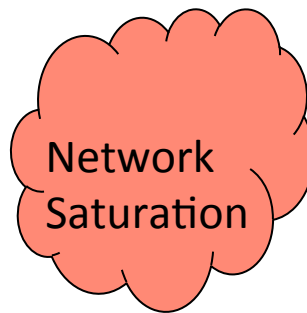
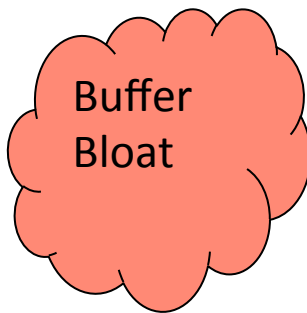


7200000
120000
2000
2

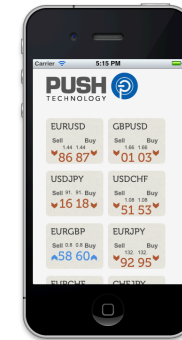
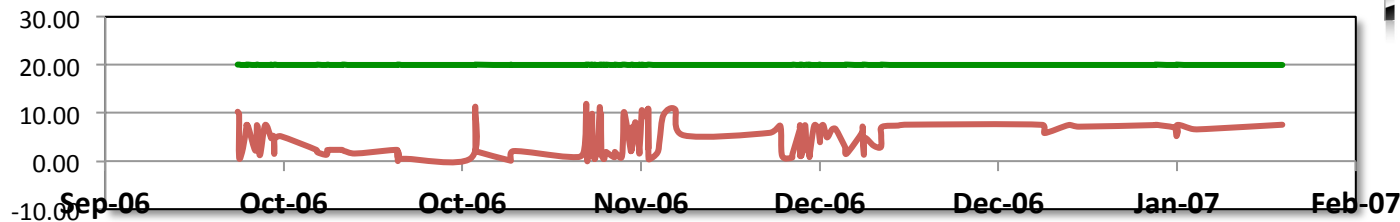
Minutes, Seconds, Millis of Internet time



The Last Mile is HARD



Download Bandwidth



ios

PushToQuote

Subscribe

Symbol	Bid Price	Ask Price	Spread	Mid
EURUSD	1.53	1.53	0.01	1.53
GBPUSD	1.73	1.82	0.09	1.78
USDJPY	99.81	98.31	1.5	99.06
USDCHF	1.13	1.12	0.01	1.12
EURGBP	0.91	0.92	0.01	0.91
USDHKD	8.71	8.4	0.31	8.55
USDDKK	5.29	5.04	0.25	5.16
EURNOK	9.16	9.09	0.07	9.13
GBPNZD	2.43	2.36	0.07	2.4
EURSEK	10.88	11.11	0.24	10.99
EURJPY	136.89	139.89	3.0	138.39
EURCHF	1.57	1.64	0.07	1.6
USDZAR	7.71	7.75	0.04	7.73
USDSEK	7.09	7.16	0.07	7.12
GBPCAD	1.74	1.8	0.05	1.77
GBPJPY	165.51	153.87	11.64	159.69
USDSGD	1.5	1.56	0.06	1.53
CHFJPY	93.77	97.97	4.2	95.87
GBPCAD	1.83	2.07	0.24	1.95
USDNOK	5.79	5.74	0.15	5.77
USDCAD	1.82	1.81	0.01	1.81
AUDUSD	0.88	0.9	0.02	0.89

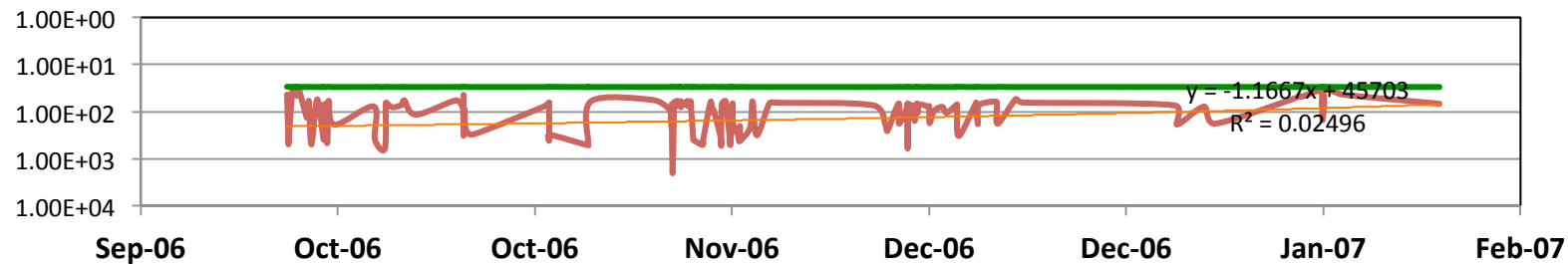
Connected to dpt://192.168.53.17:8080

Expected (Mbps)

Download (Mbps)



Latency



Latency (ms)

Expected (ms)

Linear (Latency (ms))

Conversations

M2M

Traditional Messaging
MQTT, AMQP

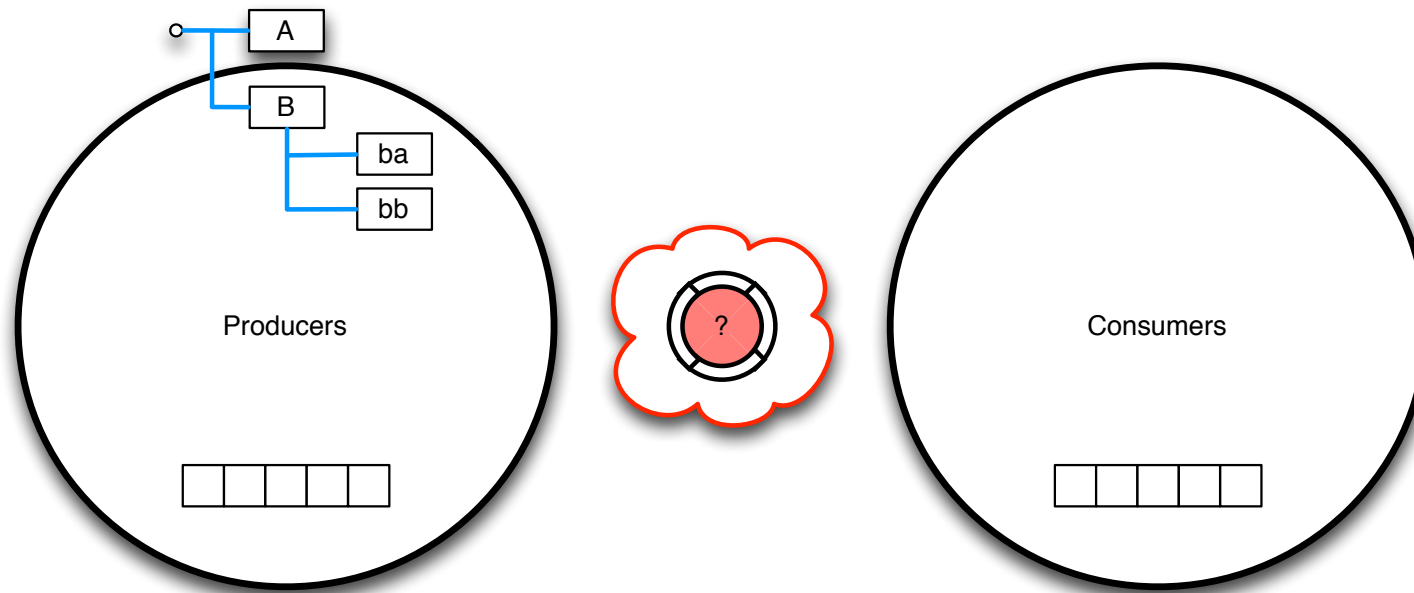
WebSockets

W3C Server Sent
Events

M2H

Bidirectional Real
Time Data Distribution

Traditional Messaging



Pros

- Loosely coupled.
- All you can eat messaging patterns
- Familiar

Cons

- No data model. Slinging **blobs**
- Fast producer, slow consumer? **Ouch.**
- No data 'smarts'. A blob is a blob.

Invented yonks ago...

Before the InterWebs

For 'reliable' networks

For machine to machine

Remember DEC Message Queues?

- That basically. Vomit!



When fallacies were simple

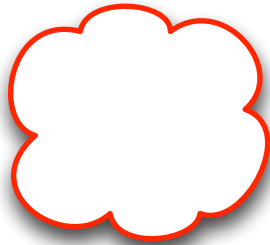
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- There is one administrator
- Topology does not change
- The network is secure
- Transport cost is zero
- The network is homogeneous

Then in 1992, this happened:

The phrase 'surfing the internet' was coined by Jean Poly.

First SMS sent

First base.



It grew, and it grew



Then in 2007, this happened:

The **god** phone:

Surfing died. Touching happened.

Second base unlocked.




Then in 2007, this happened:

So we took all the things and put them in the internet:

Cloud happened.

So we could touch all the things.

They grew and they grew like all the good things do.



Messaging
Apps
Hardware
Virtualize all the things
Services
Skills,
Specialties

Then in 2007, this happened:

And it grew and it grew.

Like all the good things do.



So we scaled all the things

Big data is fundamentally about extracting value, understanding implications, gaining insight so we can learn and improve continuously.

It's nothing new.



But, problem: The bird, basically.

Immediately Inconsistent.
But, Eventually Consistent
... Maybe.

Humans
Are
grumpy



Stop.

- The network is **not** reliable
nor is it cost free.
- Latency is **not** zero
nor is it a democracy.
- Bandwidth is **not** infinite
nor predictable especially the last mile!
- There is **not only** one administrator
trust, relationships **are** key
- Topology **does** change
It should, however, converge eventually
- The network is **not** secure
nor is the data that flows through it
- Transport cost is **not** zero
but what you **don't do** is **free**
- The network is **not** homogeneous
nor is it **smart**

Look.

- **What** and **How** are what geeks do.
- **Why** gets you paid
- **Business Value** and **Trust** dictate What and How
- Policies, Events and Content *implements **Business Value***

- ***Science** basically. But think like a carpenter:*

- *Measure twice. Cut once.*

Listen.

- **Every** nuance comes with a set of tradeoffs.
- Choosing the right ones can be hard, but it pays off.
- Context, Environment are critical
- Break all the rules, one benchmark at a time.

– Benchmark Driven Development

Action: Data Distribution

Messaging remixed around:

Relevance - Queue depth for conflatable data should be 0 or 1. No more

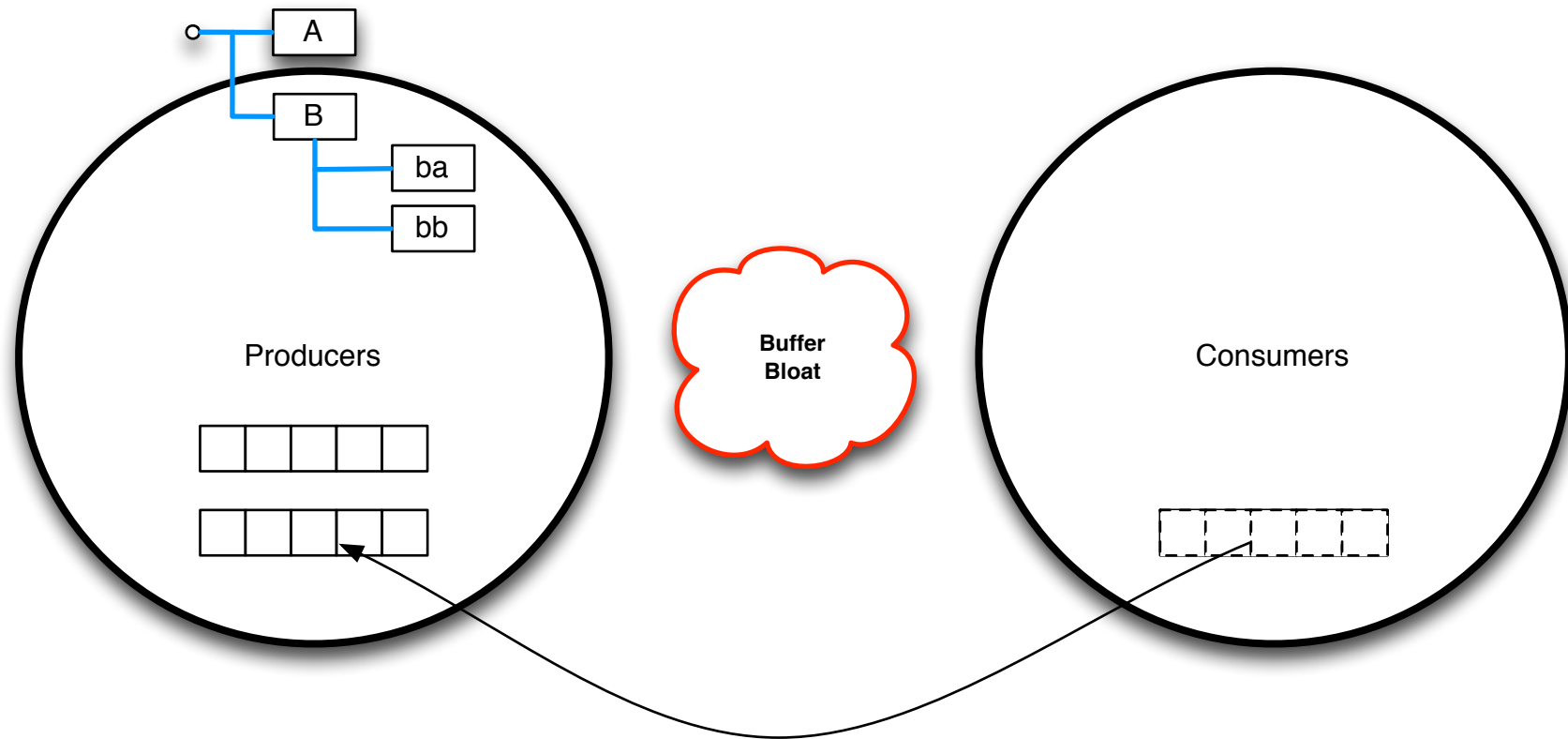
Responsiveness - Use HTTP/REST for things. Stream the little things

Timeliness - It's relative. M2M != M2H.

Context - Packed binary, deltas mostly, snapshot on subscribe.

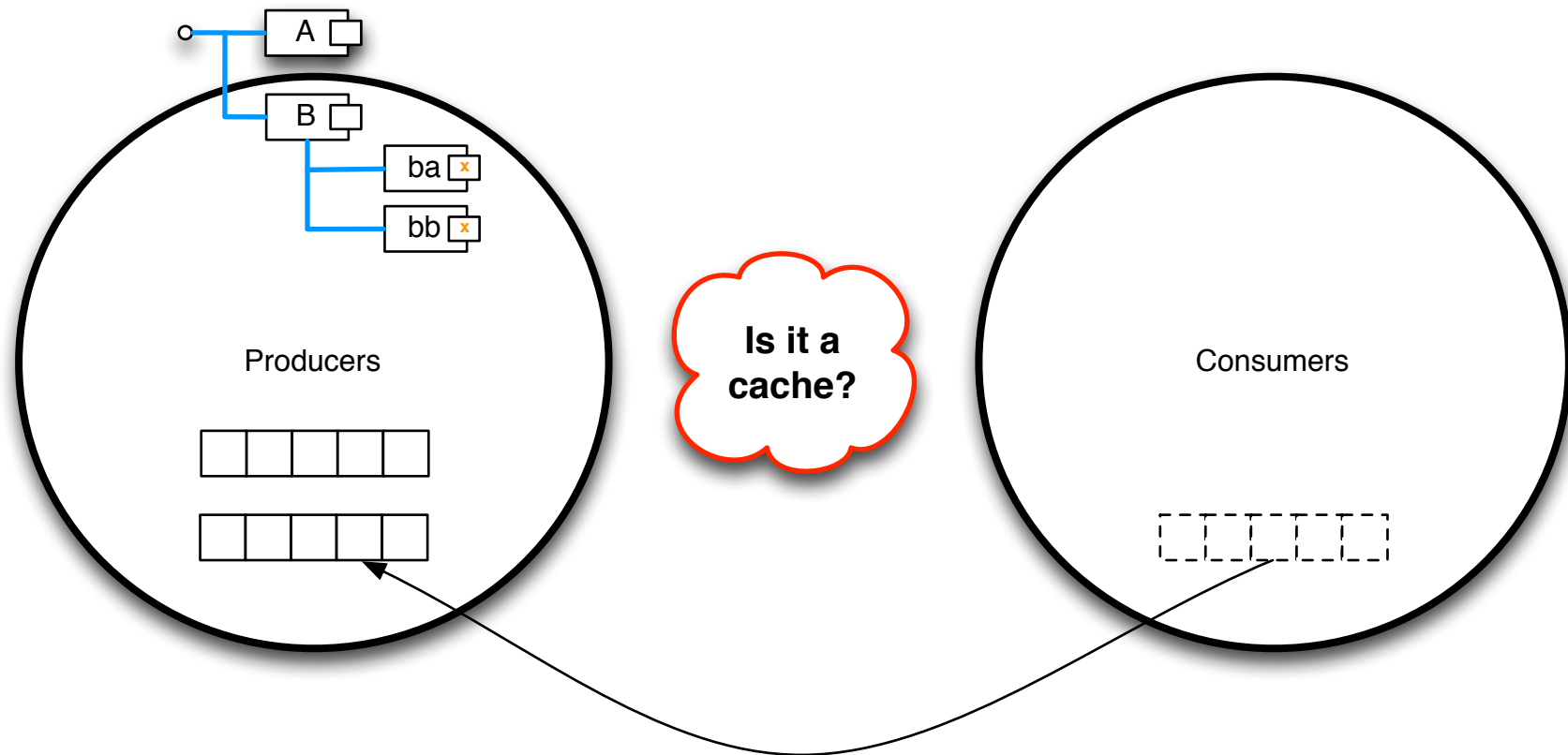
Environment- Don't send 1M 1K events to a mobile phone with 0.5mbps.

Action. Virtualize Client Queues



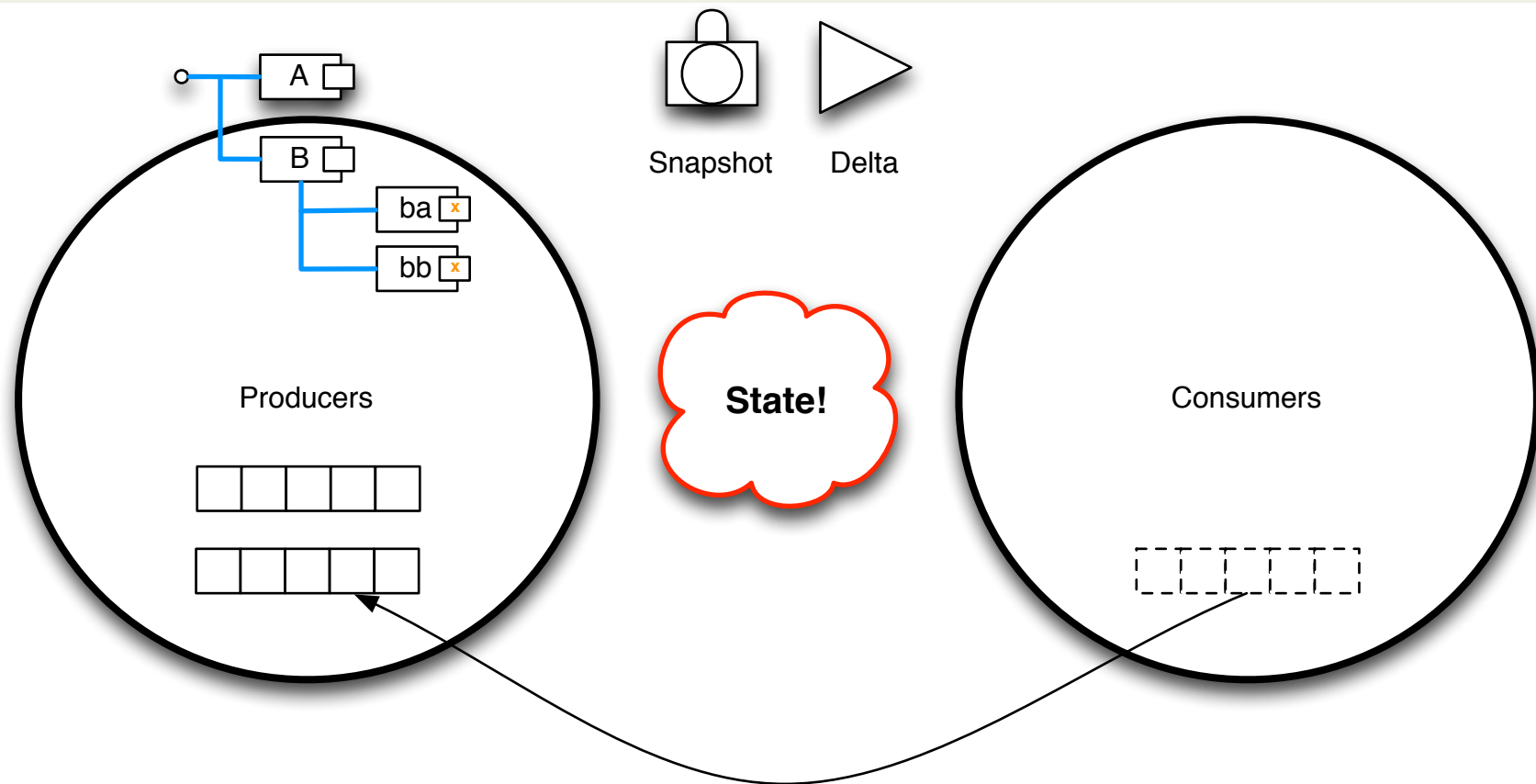
Nuance: Client telemetry. Tradeoff: Durable subscriptions harder

Action. Add data caching



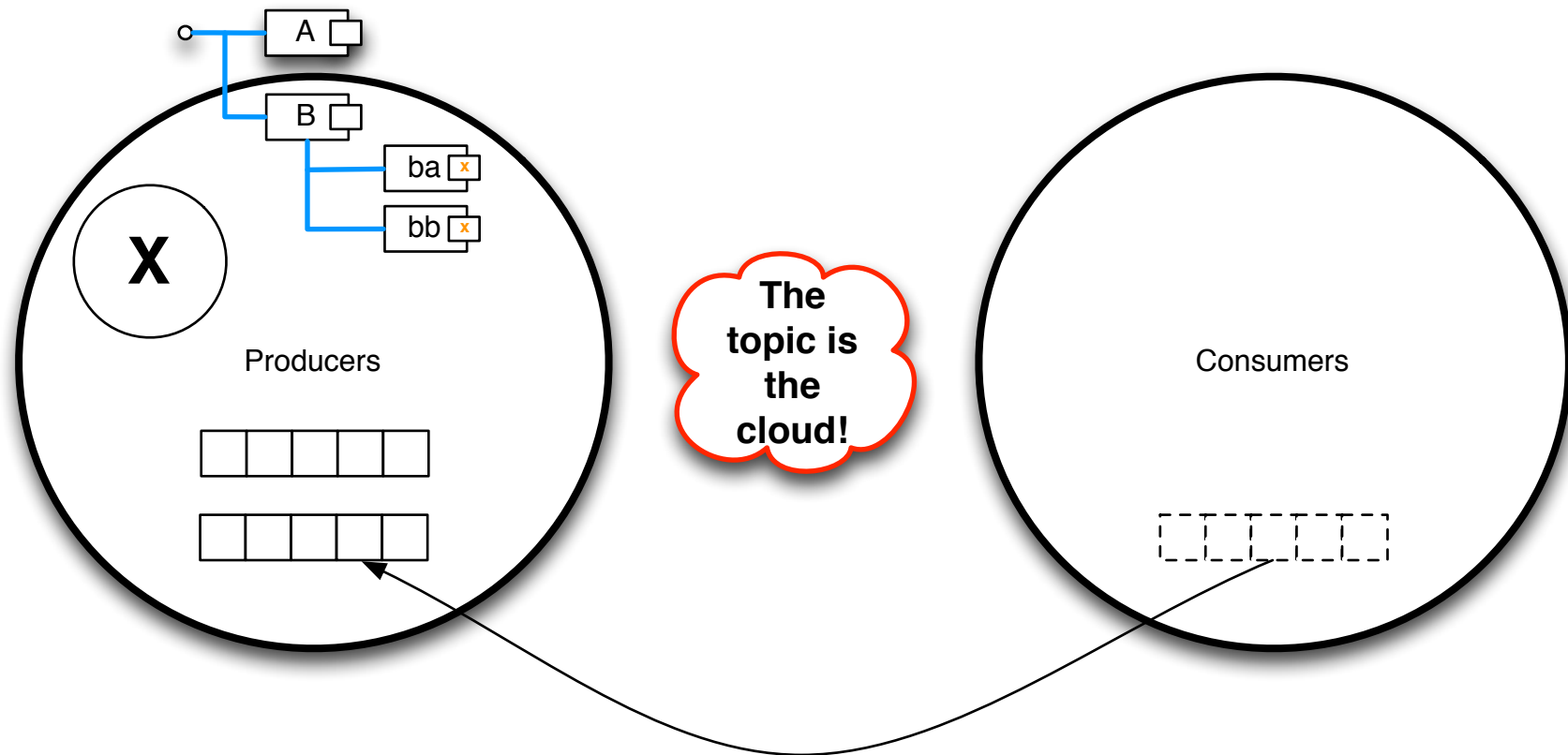
One hop closer to the edge ...

Action. Exploit data structure



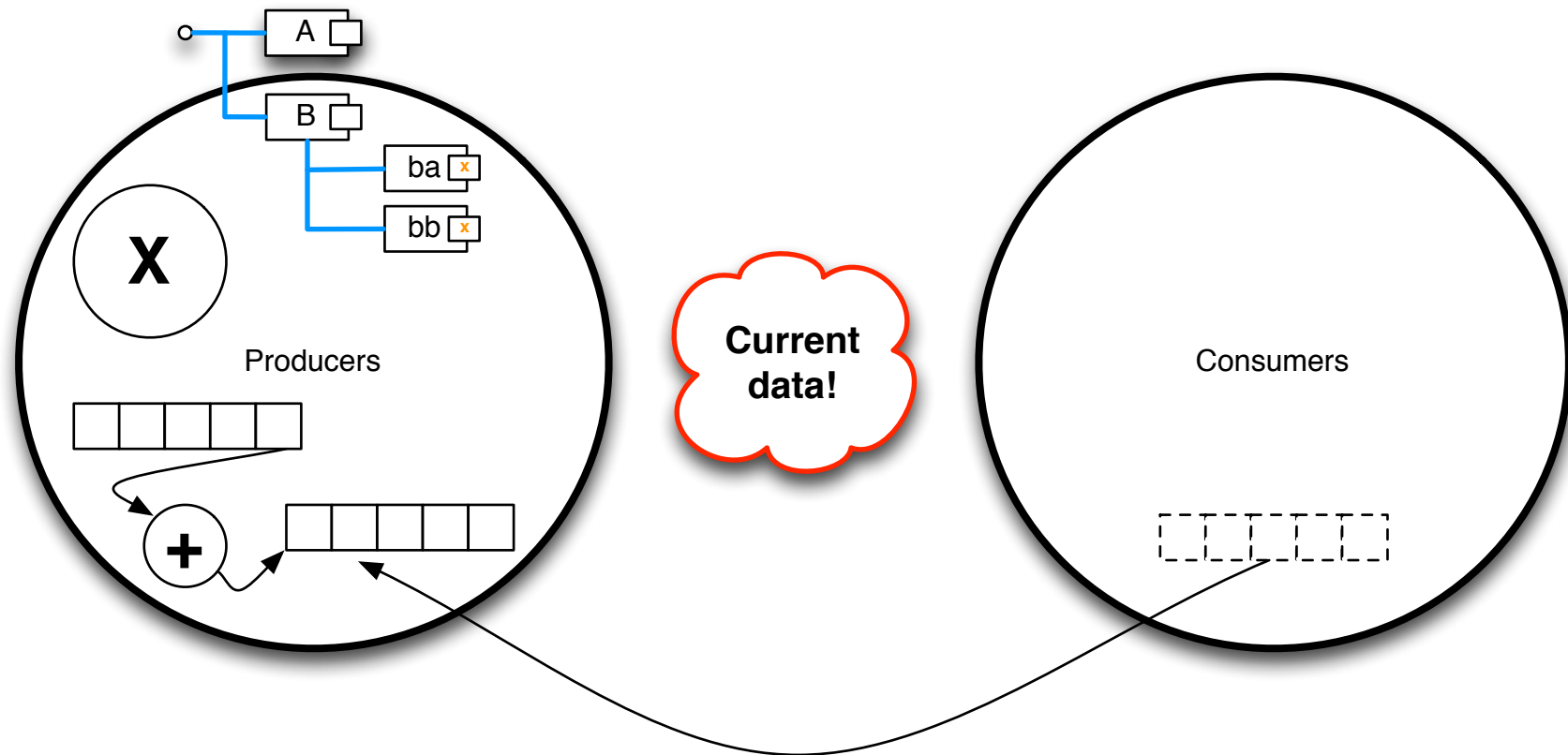
Snapshot recovery. Deltas or Changes mostly. Conserves bandwidth

Action. Behaviors



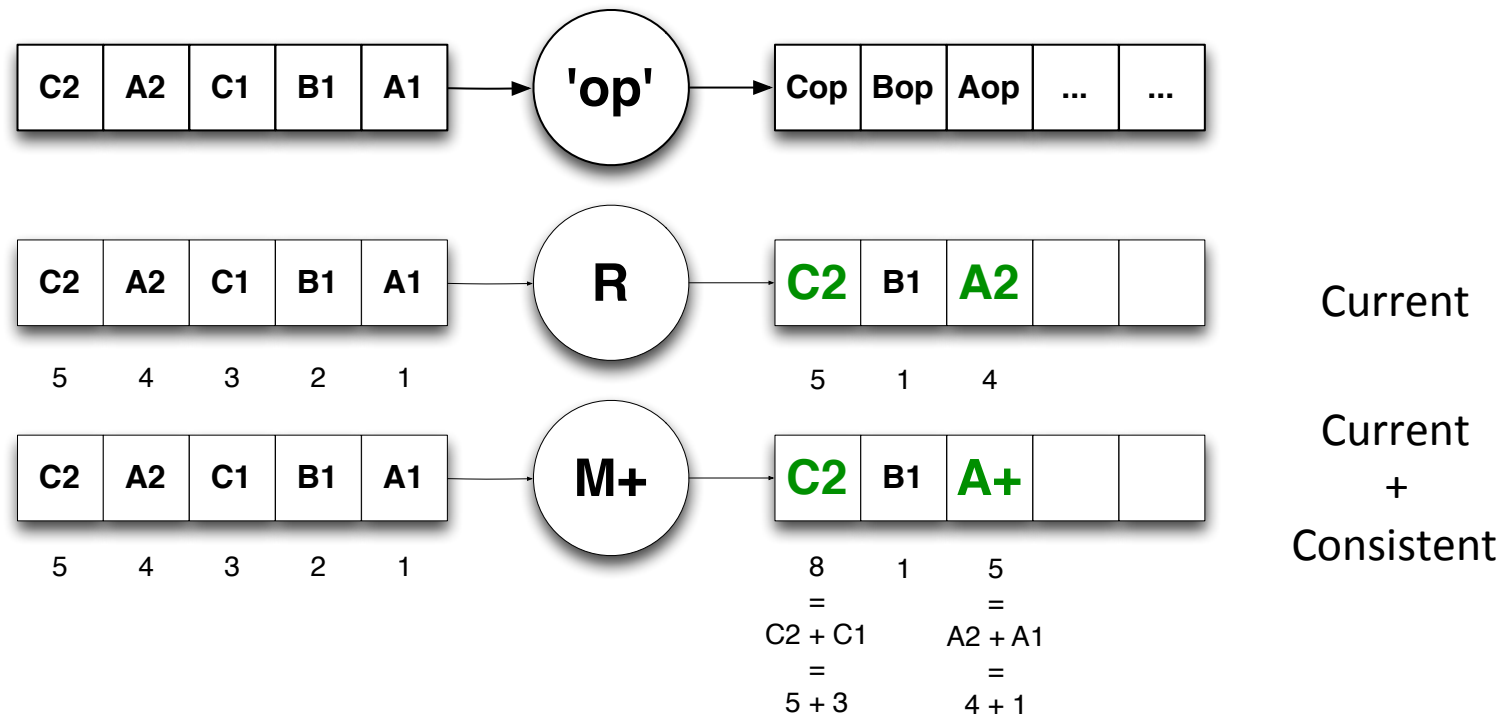
Extensible. Nuance? Roll your own protocols. Tradeoff? 3rd party code in the engine :/

Action. Structural Conflation



Ensures only current + consistent data is distributed. Actively soaks up bloat. Extensible!

Action. Structural Conflation [EEP]



Replace

- Replace/Overwrite 'A1' with 'A2'
- 'Current data right now'
- Fast

Merge

- Merge A1, A2 under some operation.
 - Operation is pluggable
 - Tunable consistency
 - Performance $f(\text{operation})$

Implement *Erlang* Client ☺

```
main_loop(Pid,Count) ->
  receive
    {snapshot, Topic , _Headers , Body } -> io:format("Snapshot: ~s ~s~n", [ Topic , Body]);
    {delta, Topic , _Headers, Body } -> io:format("Delta: ~s ~s~n", [ Topic , Body]);
    {ping, Timestamp} -> io:format("Ping: ~w ~n", [ Timestamp ]);
    {topic_status, Topic, Status} -> io:format("Topic Status: ~s ~s~n", [ Topic, Status]);
    What -> io:format("Unexpected: ~w ~n", [What])
  after 10 -> ok
end,
main_loop(Pid,Count+1).
```


Example Diffusion *Erlang* Client

```
handle_event({log, Message}, #diffusion_lager_be_state{level = _L,
    formatter = Formatter,
    format_config = FormatConfig } = State) ->
    Msg = Formatter:format(Message, FormatConfig),
    {ok, push(State, Msg)};

push(State, Msg) ->
    T = erlang:list_to_binary(State#diffusion_lager_be_state.topic),
    R = io_lib:format("~s", [Msg]),
    S = erlang:list_to_binary(lists:flatten(R)),
    pusherl_client:send(State#diffusion_lager_be_state.cli, T , << S/binary >> ),
    State.
```

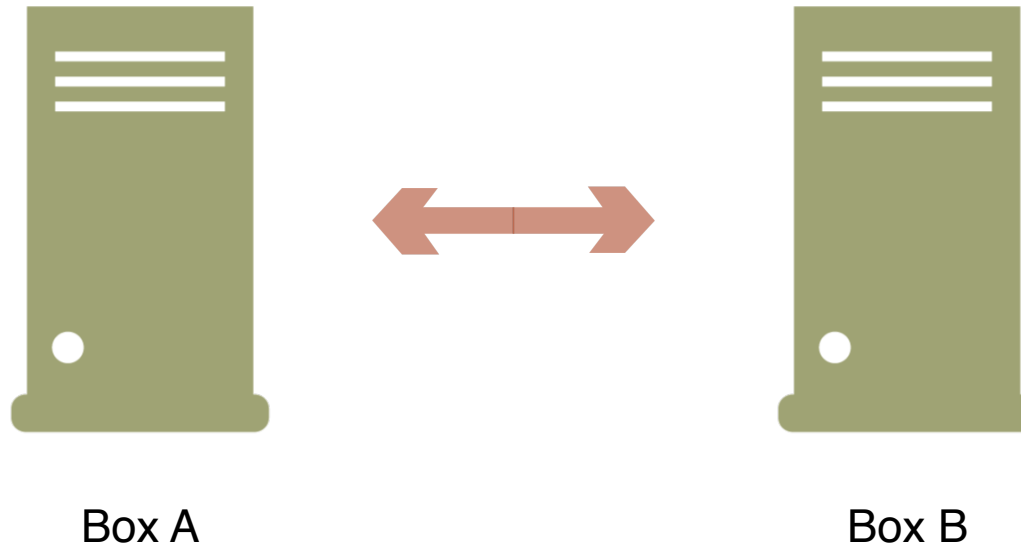
Backend to Basho's Lager logging – Stream log events for near real-time analysis ...



Performance is **not** a number



Benchmark Models



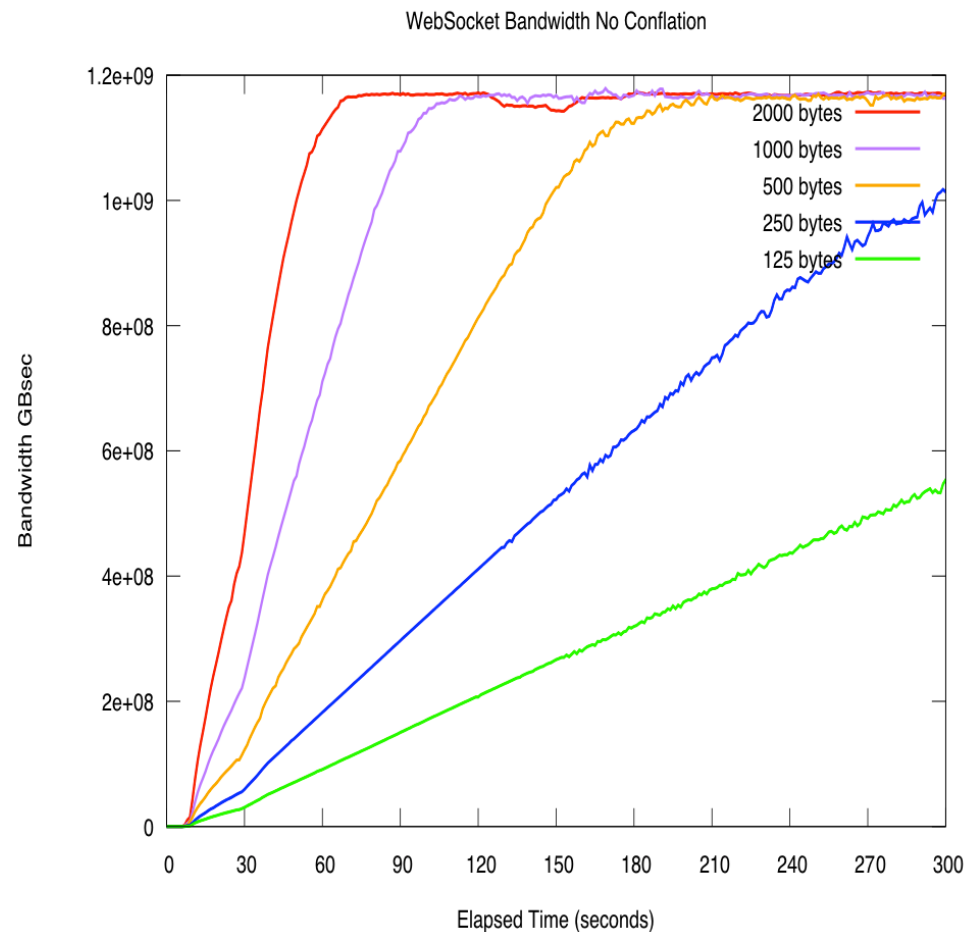
Throughput (Worst case)

- Ramp clients continuously
- 100 messages per second per client
- Payload: 125 .. 2000 bytes
- Message style vs with conflation

Latency (Best case)

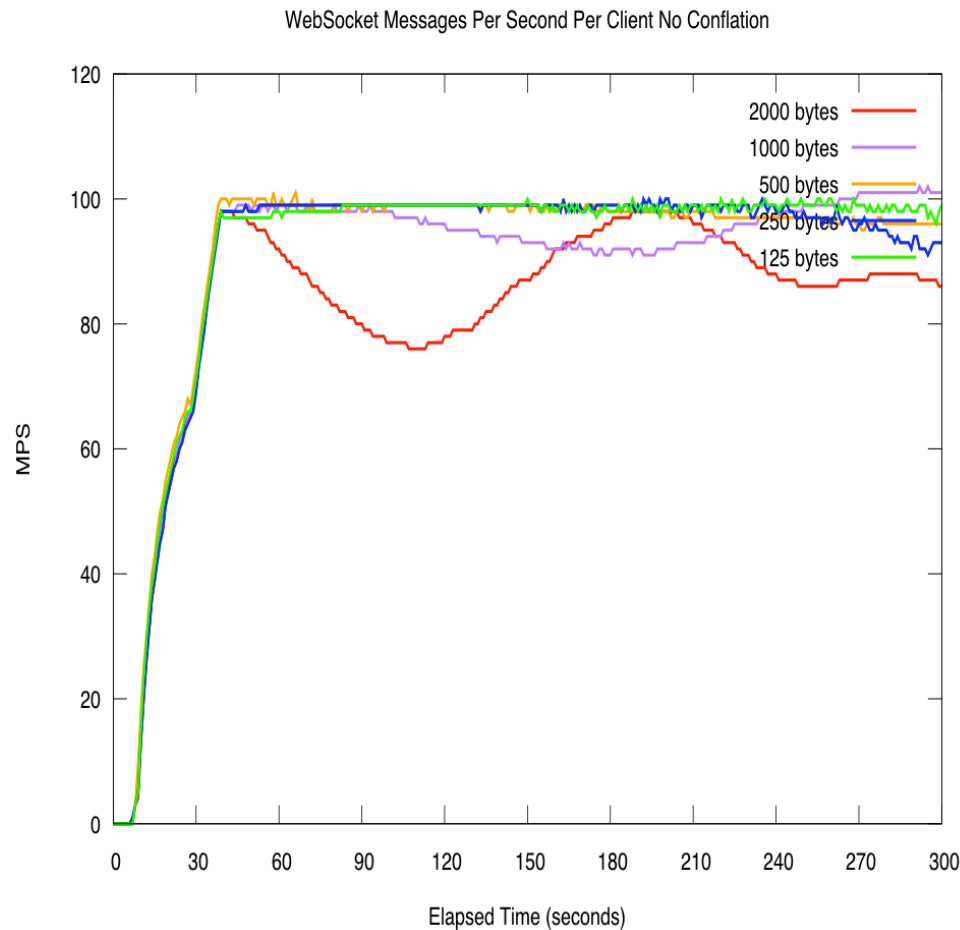
- Really simple. 1 client
- Ping – Pong. Measure RTT time.
- Payload: 125 .. 2000 bytes

Throughput. Non-conflated



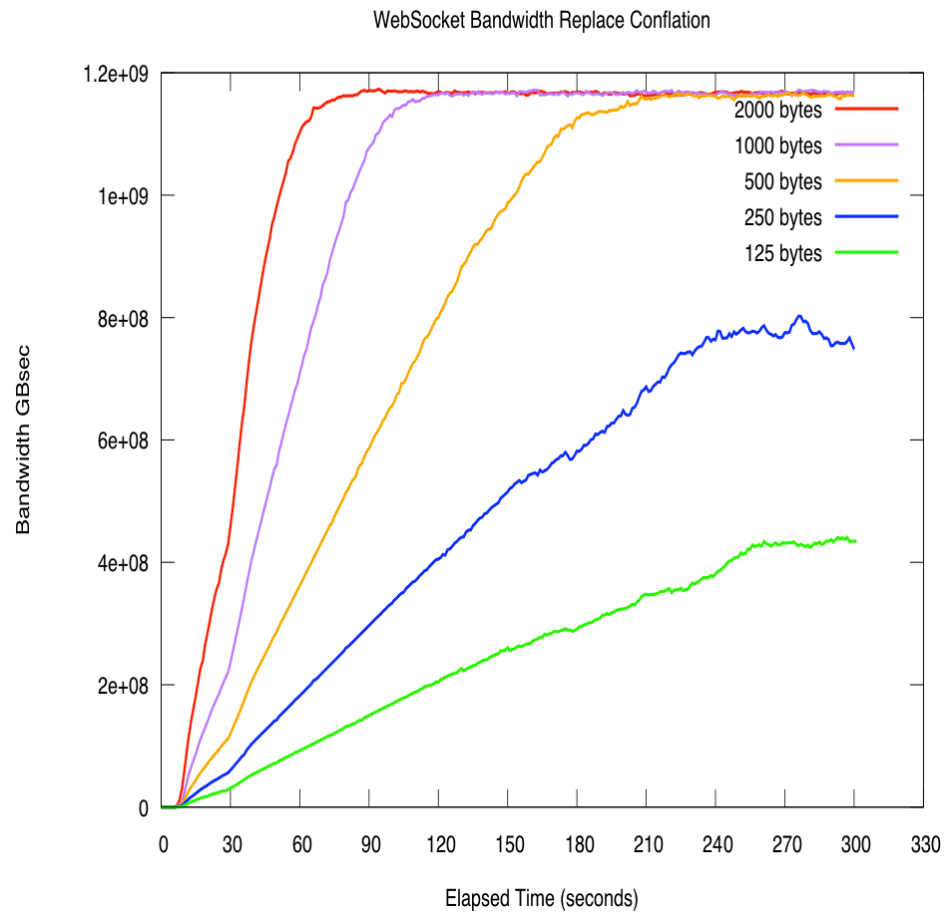
- Cold start server
- Ramp 750 clients 'simultaneously' at 5 second intervals
- 5 minute benchmark duration
- Clients onboarded linearly until IO (here) or compute saturation occurs.
- What the 'server' sees

Throughput. Non-conflated



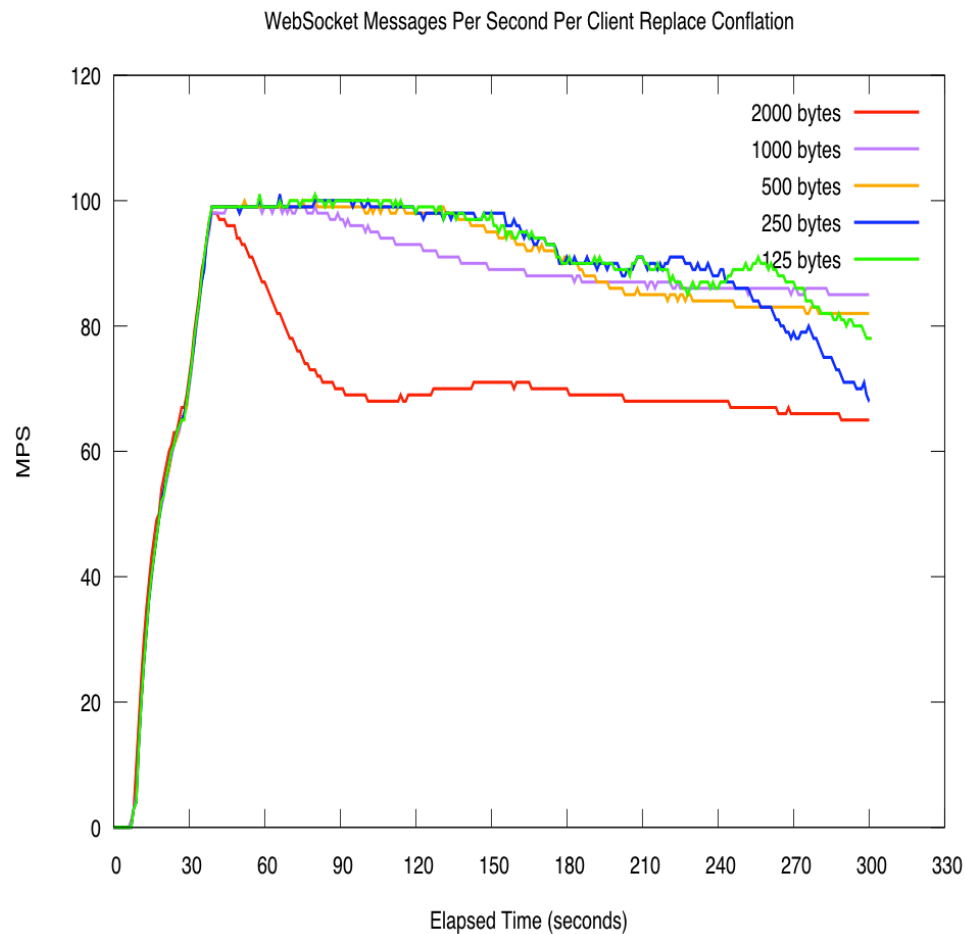
- What the 'client' sees
- At and beyond saturation of some resource?
- Things break!
- New connections fail. Good.
- Long established connections ok.
- Recent connections timeout and client connections are dropped. Good.
- Diffusion handles smaller message sizes more gracefully
- Back-pressure 'waveform' can be tuned out. Or, you could use structural conflation!

Throughput. Replace-conflated



- Cold start server
- Ramp 750 clients 'simultaneously' at 5 second intervals
- 5 minute benchmark duration
- Clients onboarded linearly until IO (here) or compute saturation occurs.
- Takes longer to saturate than non-conflated case
- Handles more concurrent connections
- Again, 'server' view

Throughput. Replace-Conflated

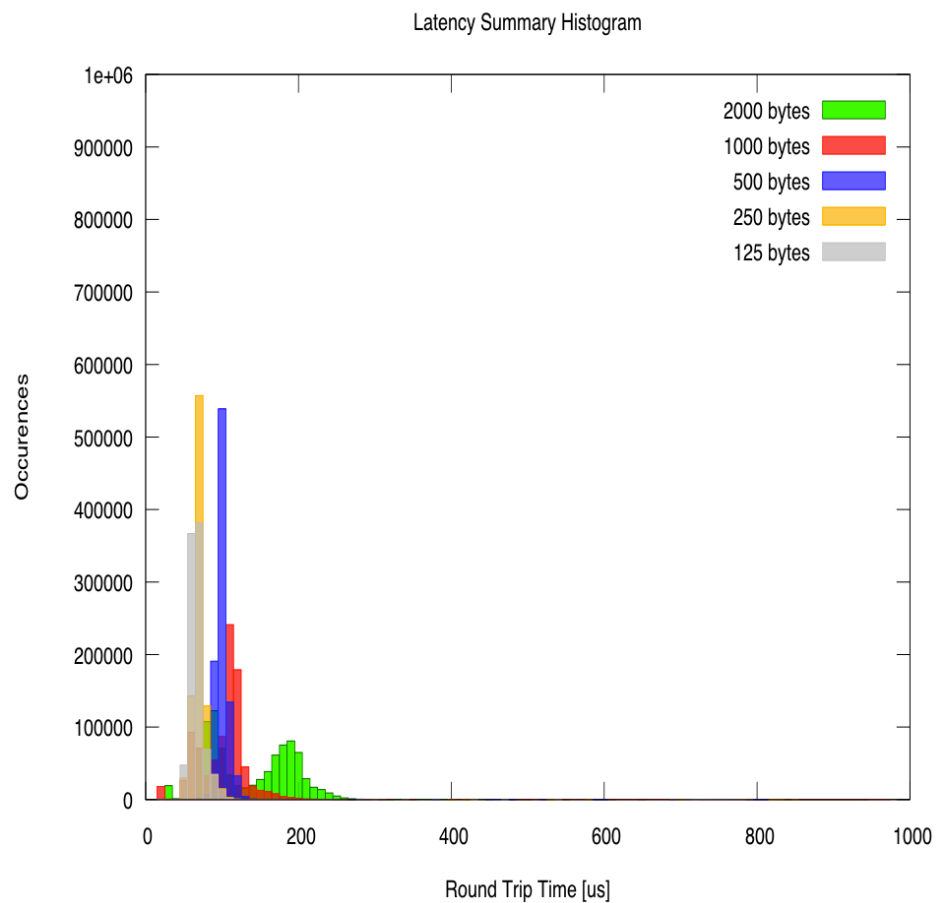


- What the 'client' sees
- Once saturation occurs Diffusion adapts actively by degrading messages per second per client
- This is good. Soak up the peaks through fairer distribution of data.
- Handles spikes in number of connections, volume of data or saturation of other resources using a single load adaptive mechanism

Throughput. Stats

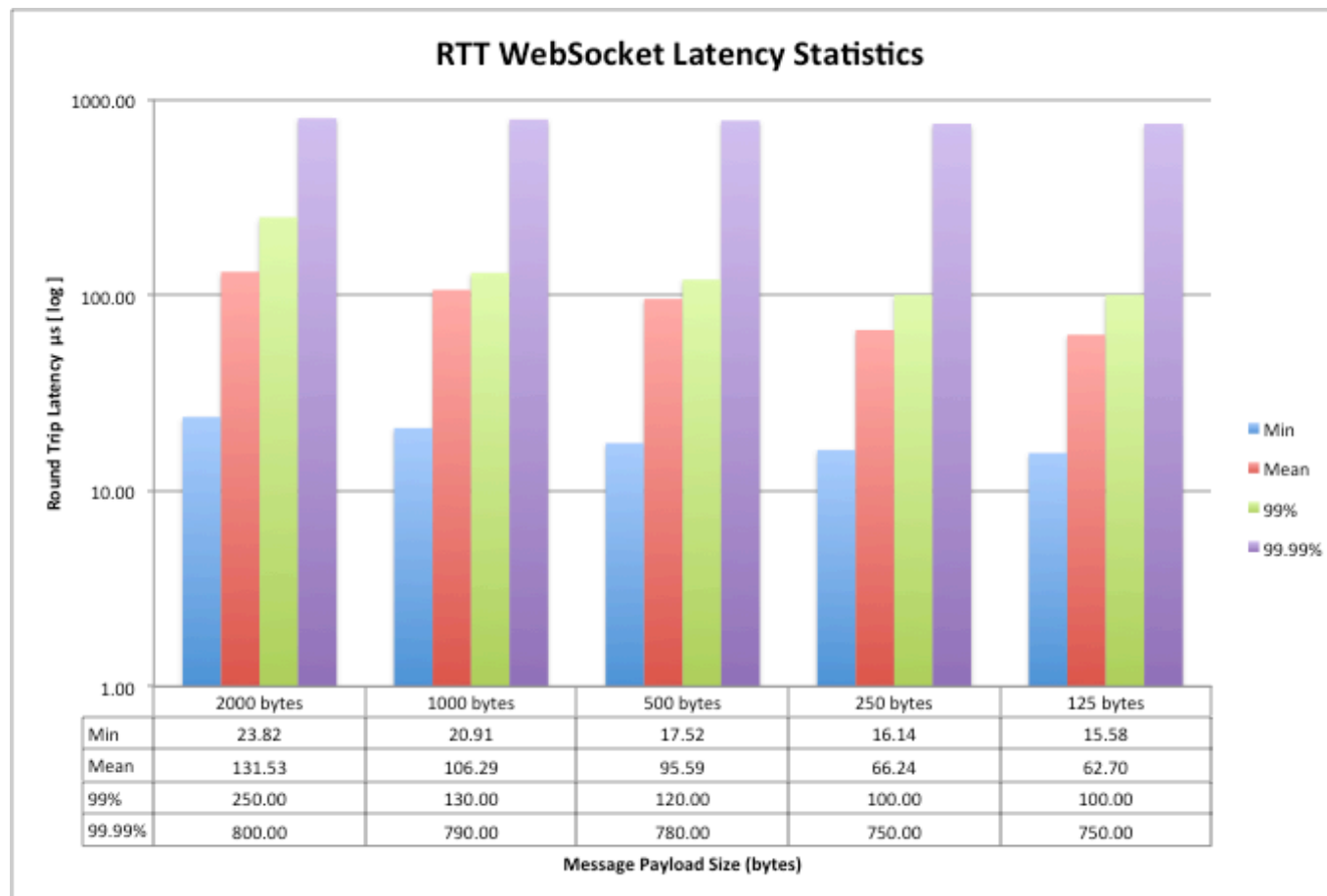
- Data / (Data + Overhead)?
 - 1.09 GB/Sec payload at saturation.
 - 10Ge offers theoretic of 1.25GB/Sec.
 - Ballpark 87.2% of traffic represents business data.
- Benefits?
 - Optimize for business value of distributed data.
 - Most real-world high-frequency real-time data is recoverable
 - Stock prices, Gaming odds
 - Don't use conflation for transactional data, natch!

Latency (Best case)



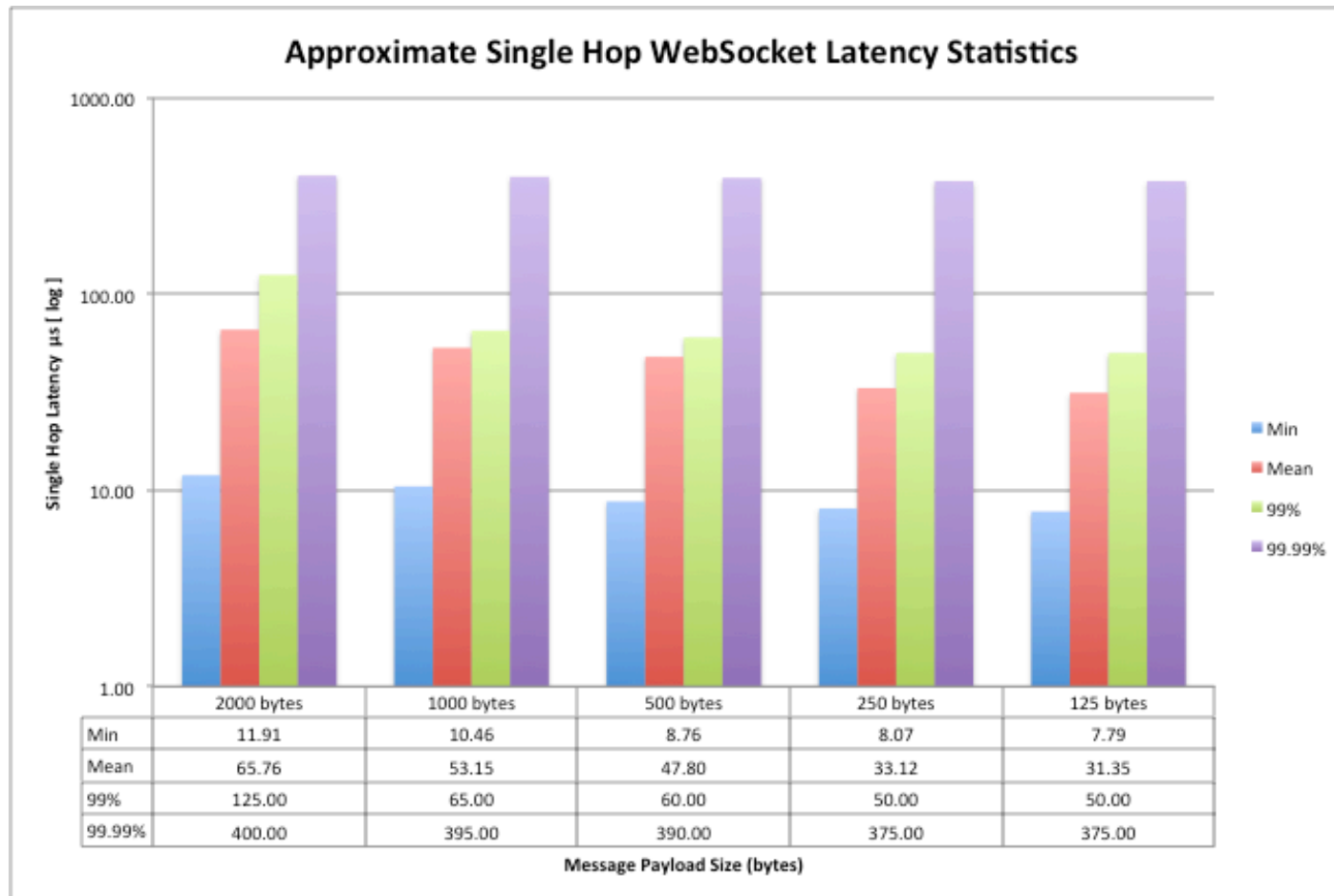
- Cold start server
- 1 solitary client
- 5 minute benchmark duration
- Measure ping-pong round trip time
- Results vary by network card make, model, OS and Diffusion tuning.

Latency. Round Trip Stats



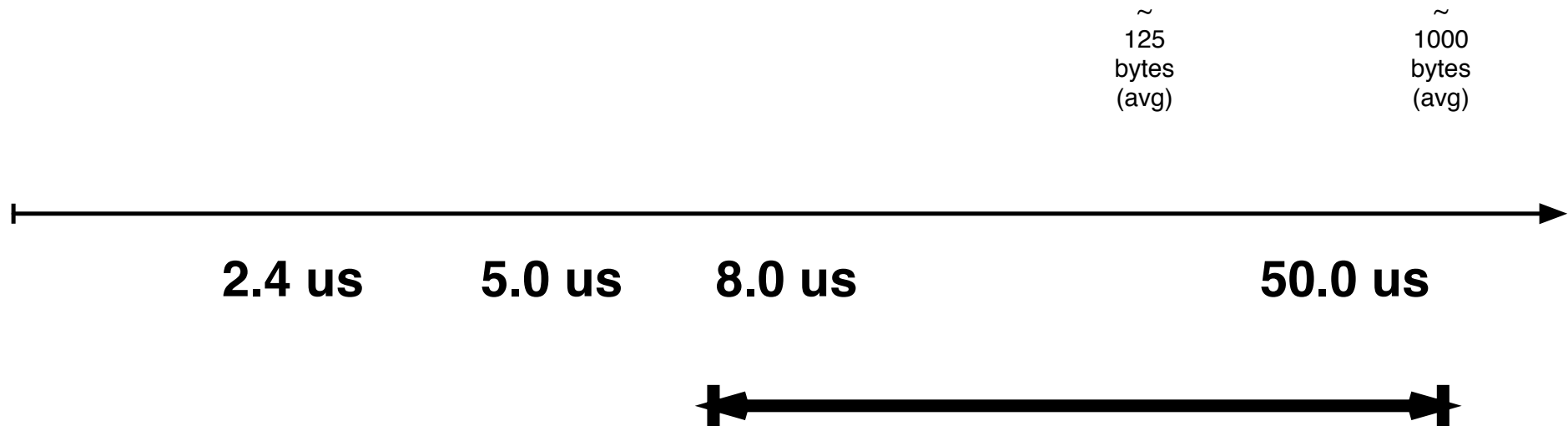
- Sub-millisecond at the 99.99% percentile
- As low as 15 microseconds
- On average significantly sub 100 microseconds for small data

Latency. Single Hop Stats

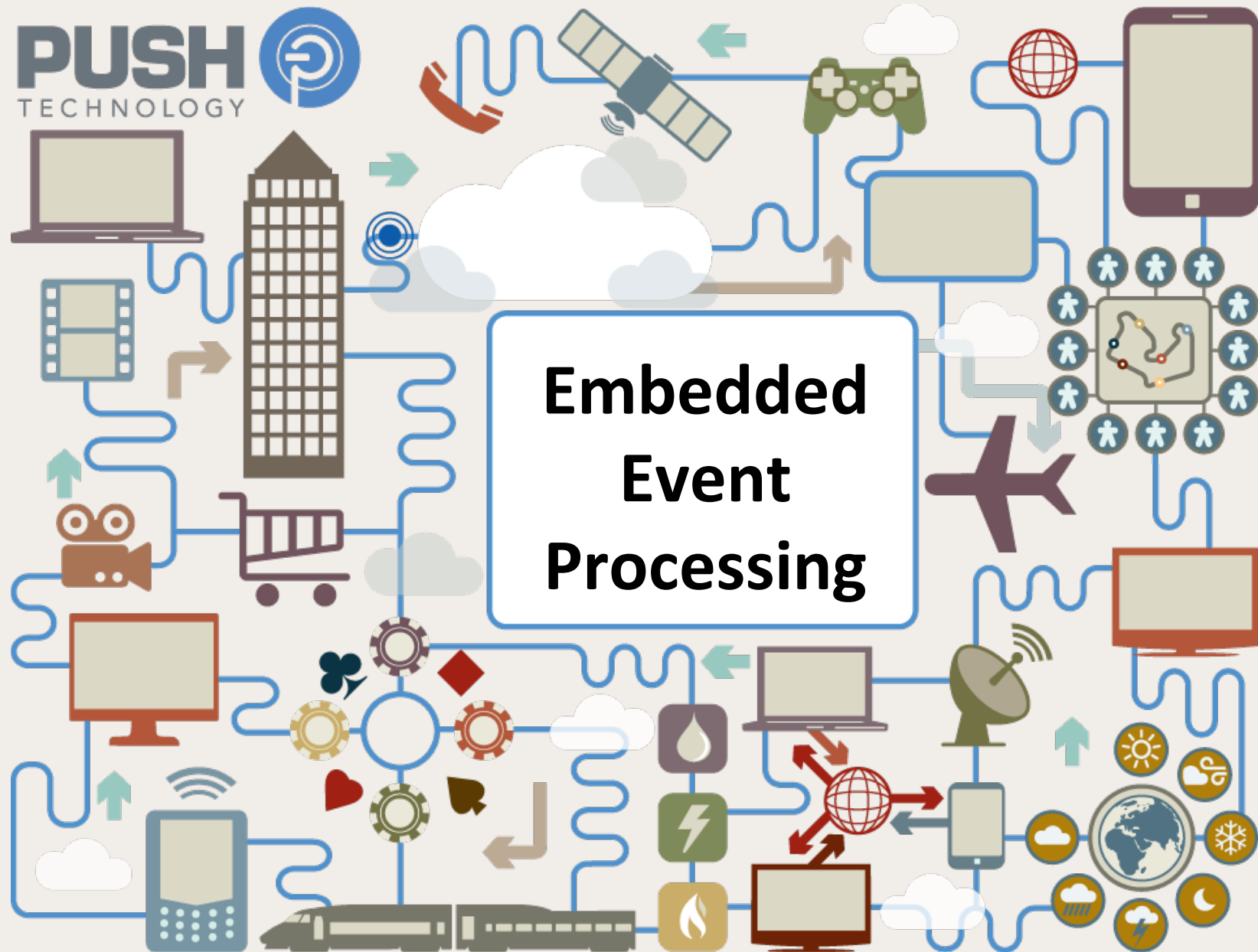


- Sub 500 microseconds at the 99.99% percentile
- As low as 8 microseconds
- On average significantly sub 50 microseconds for small data

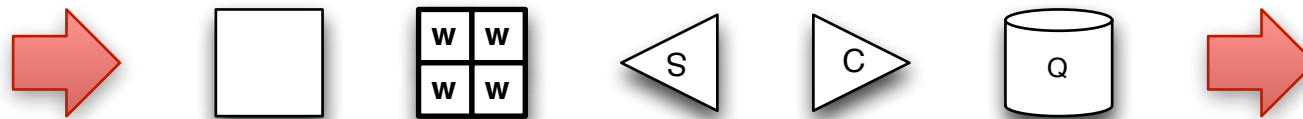
Latency in perspective



- 2.4 us. A tuned benchmark in C with low latency 10Ge NIC, with kernel bypass, with FPGA acceleration
- 5.0 us. A basic java benchmark – as good as it gets in java
- Diffusion is measurably ‘moving to the left’ release on release
- We’ve been actively tracking and continuously improving since 4.0



CEP, basically



What is eep.erl?

- Add aggregate functions and window operations to Erlang
- Separate context (window) from computation (aggregate fn)
- 4 window types: tumbling, sliding, periodic, monotonic
- Process oriented.
- Fast enough. 😊

Aggregate Functions

```
-module(eep_aggregate).  
  
-export([behaviour_info/1]).  
  
behaviour_info(callbacks) ->  
    [ { init, 0} , { accumulate, 2}, {compensate, 2}, {emit, 1} ];  
  
behaviour_info(_) ->  
    undefined.
```

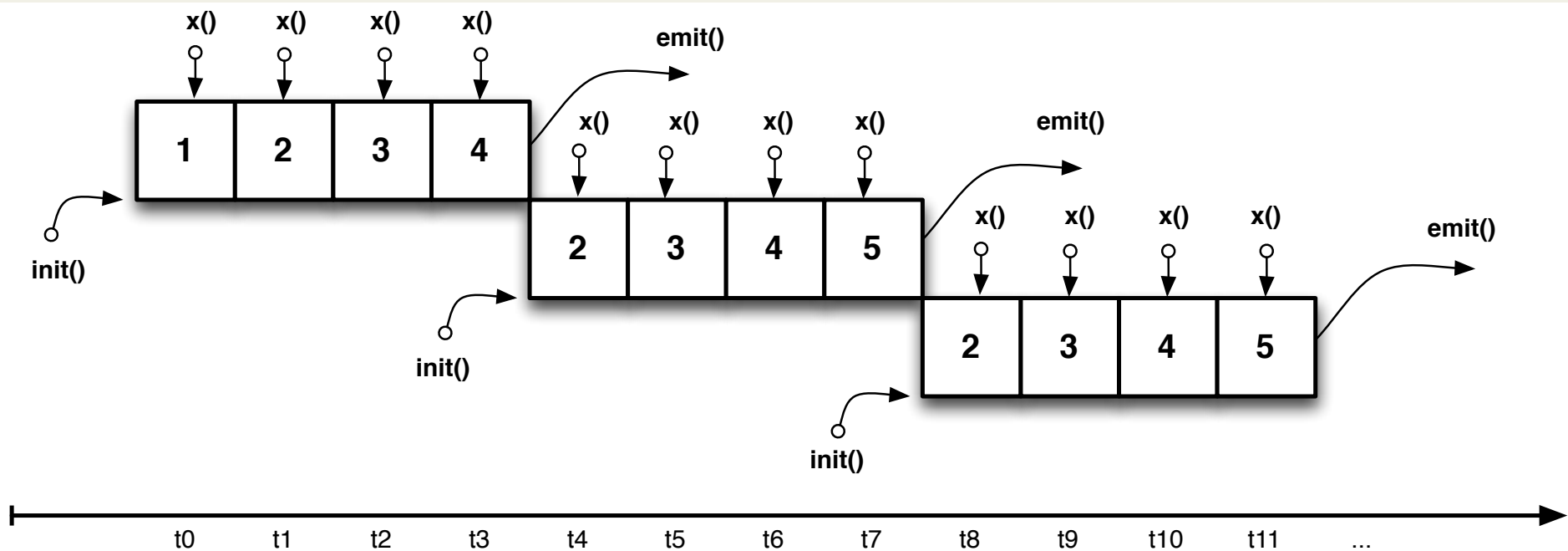
What is an aggregate function?

- A function that computes values over events.
- The cost of calculations are amortized per event
- Just follow the above recipe
- Example: Aggregate 2M events (equity prices), send to GPU on emit, receive 2M options put/call prices as a result.

Aggregate Function Example

```
%% aggregate behaviour callbacks.  
-export([init/0]).  
-export([accumulate/2]).  
-export([compensate/2]).  
-export([emit/1]).  
  
init() ->  
    0.  
  
accumulate(State,X) ->  
    case State >= X of true -> State; false -> X end.  
  
compensate(State,X) ->  
    case State >= X of true -> X; false -> State end.  
  
emit(State) ->  
    State.
```

Tumbling Windows



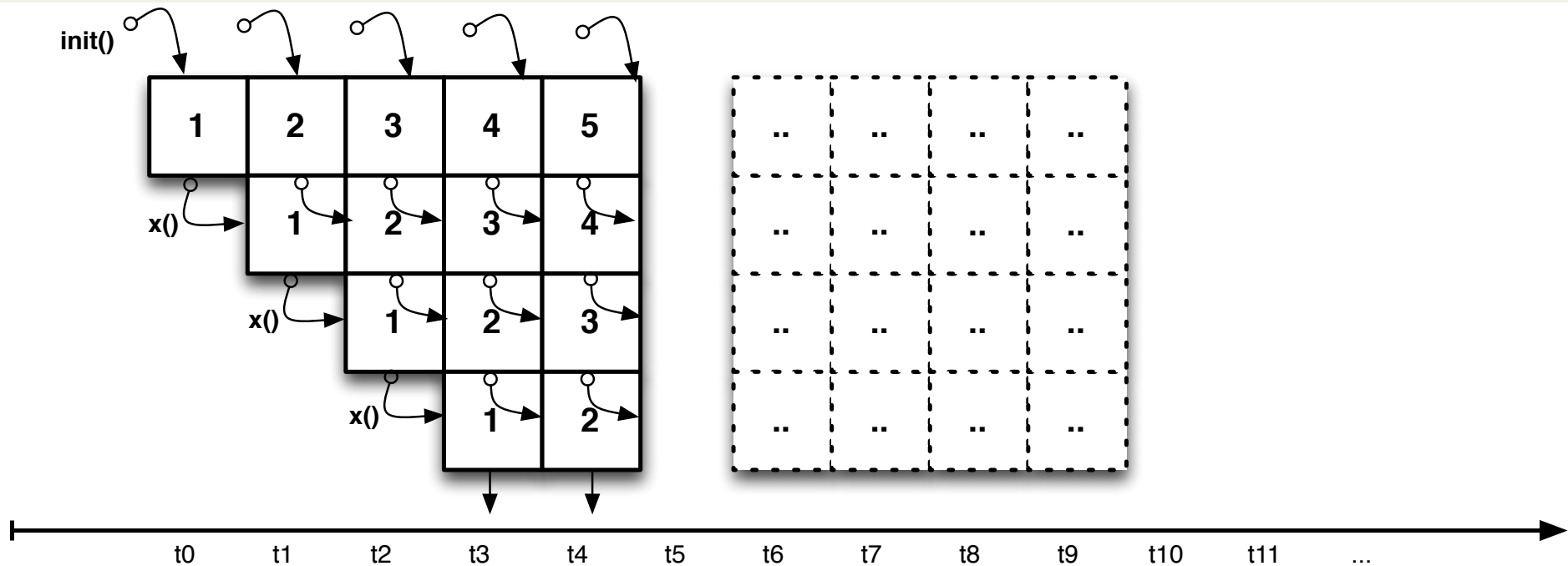
What is a tumbling window?

- Every N events, give me an average of the last N events
- Does not overlap windows
- 'Closing' a window, 'Emits' a result (the average)
- Closing a window, Opens a new window

Tumbling Windows

```
Eshell V5.9.3 (abort with ^G)
1> P = eep_window_tumbling:start(eep_stats_sum,4).
<0.34.0>
2> P ! {add_handler, eep_emit_trace, []}.
{add_handler,eep_emit_trace,[]}
3> [ P ! {push, X} || X <- lists:seq(1,24)], ok.
Emit: 10
ok
Emit: 26
Emit: 42
Emit: 58
Emit: 74
Emit: 90
4> █
```

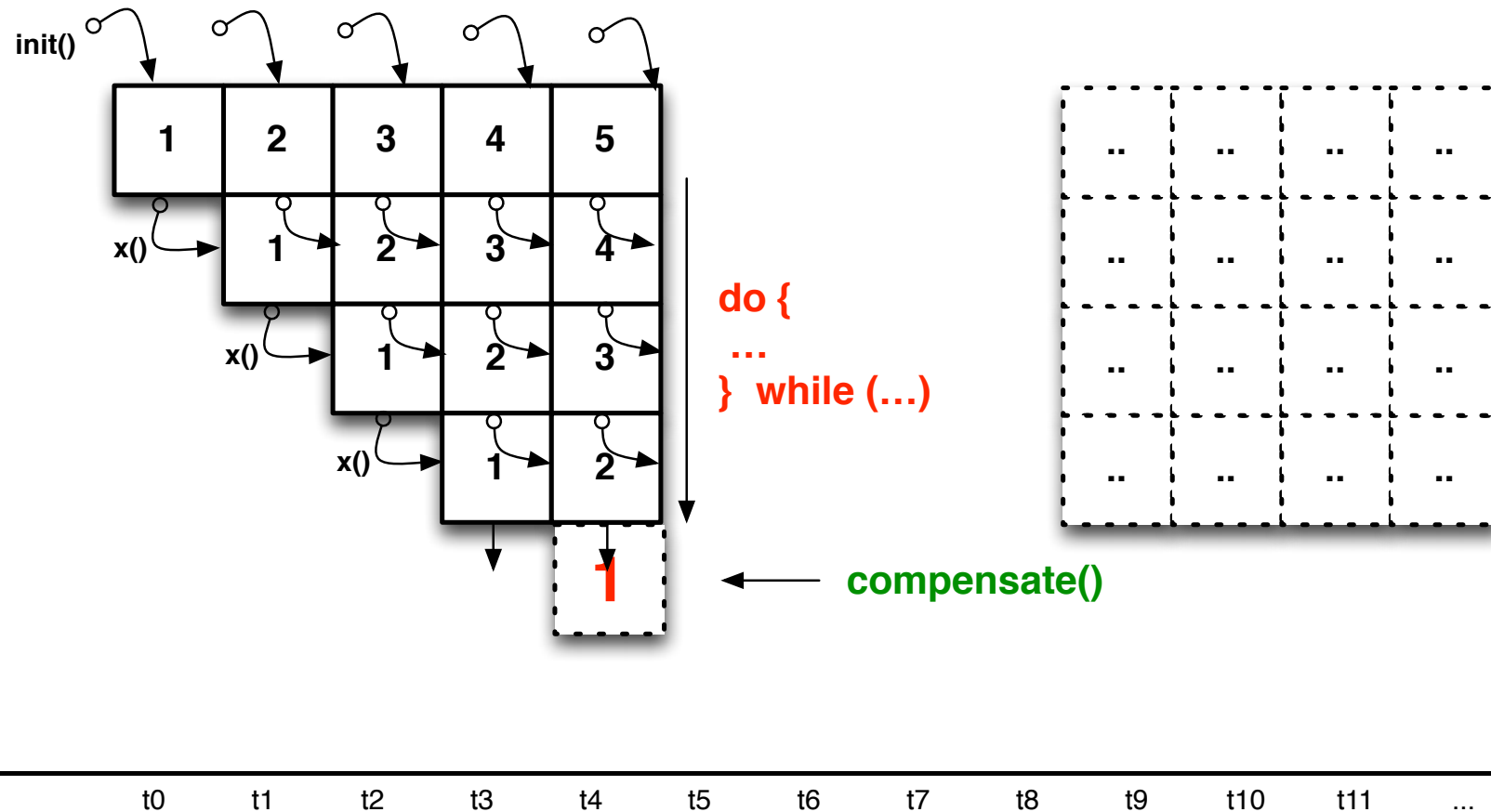
Sliding Windows - O(Long Story)



What is a sliding window?

- Like tumbling, except can overlap. But $O(N^2)$, Keep N small.
- Every event opens a new window.
- After N events, every subsequent event emits a result.
- Like all windows, cost of calculation amortized over events

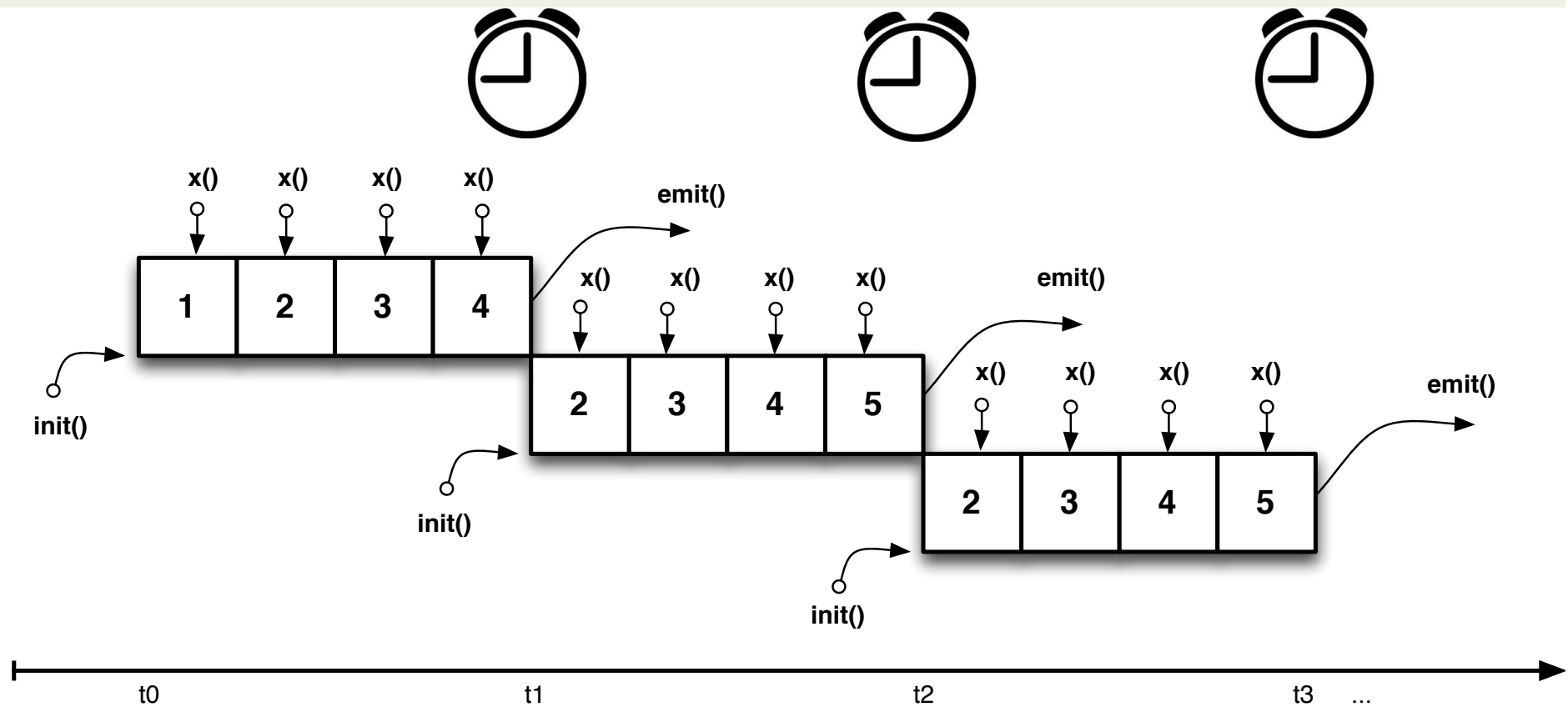
Compensating Aggregates



Sliding Windows.

```
5> f(P).  
ok  
6> P = eep_window_sliding:start(eep_stats_sum,4).  
<0.41.0>  
7> P ! {add_handler, eep_emit_trace, []}.  
{add_handler,eep_emit_trace,[]}  
8> [ P ! {push, X} || X <- lists:seq(1,9)], ok.  
Emit: 10  
ok  
Emit: 14  
Emit: 18  
Emit: 22  
Emit: 26  
Emit: 30
```

Periodic Windows



What is a periodic window?

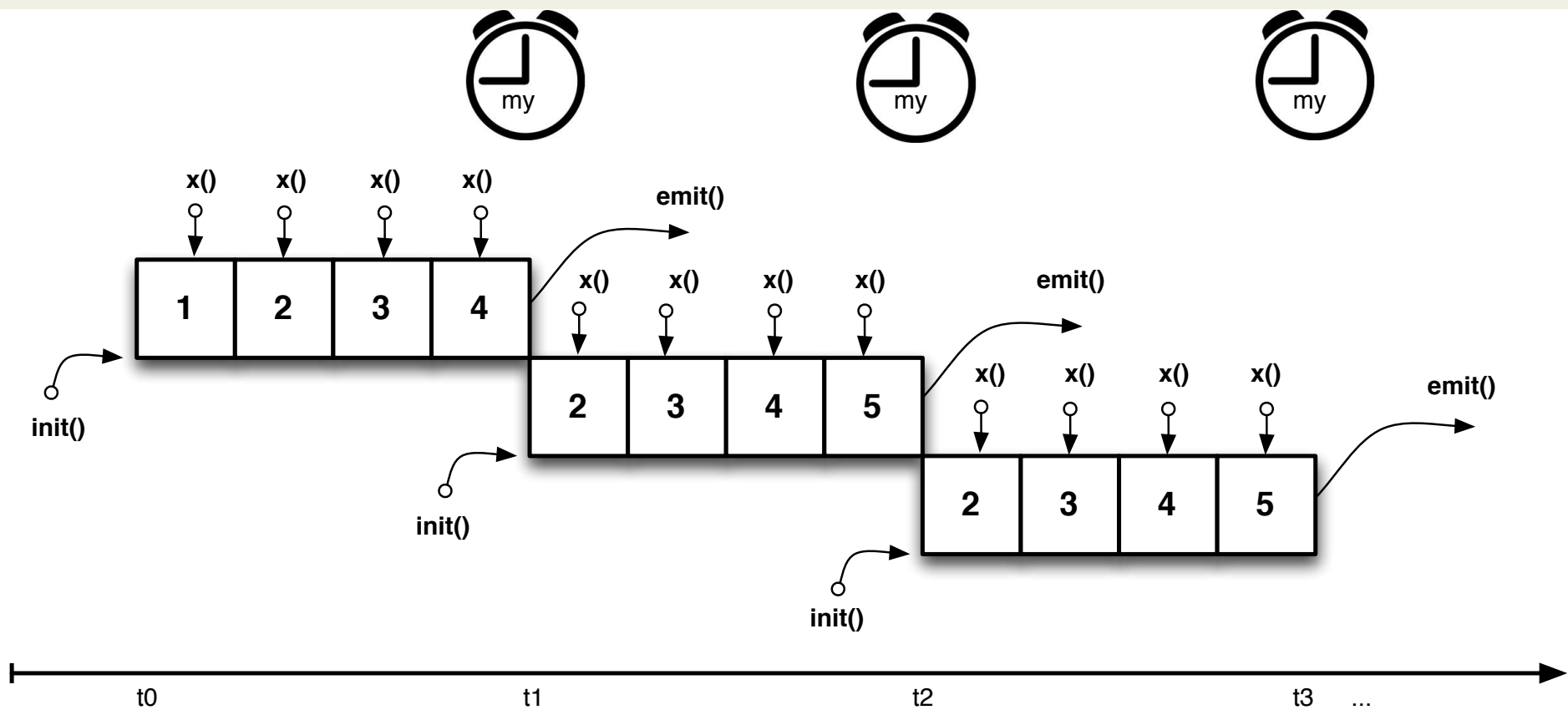
- Driven by 'wall clock time' in milliseconds
- Not monotonic, natch. Beware of NTP

Periodic Windows

```
main(_Args) ->
  P = eep_window_periodic:start(eep_stats_count,1000),
  P ! {add_handler, eep_emit_trace, []},
  P ! tick,
  main_loop(P,1).

main_loop(Window,Count) ->
  %timer:sleep(100),
  Window ! {push, 1},
  Window ! tick,
  io:format("Count: ~W~n", [Count]),
  main_loop(Window,Count+1).
```

Monotonic Windows



What is a monotonic window?

- Driven mad by 'wall clock time'? Need a logical clock?
- No worries. Provide your own clock! Eg. Vector clock

Monotonic Windows

```
export C[main/1]}.

main(_Args) ->
  P = eep_window_monotonic:start(eep_stats_count,eep_clock_count,1000),
  P ! {add_handler, eep_emit_trace, []},
  main_loop(P,1).

main_loop(Window,Count) ->
  Window ! {push, 1},
  Window ! tick,
  main_loop(Window,Count+1).
```

EEP looking forward?

- Migrate windows from simple processes to `gen_server`.
- Extract library (no process / message passing overhead).
- Consider NIFs for performance critical sections.
- Consider native aggregate functions.
- Consider alternative to `gen_event` for interfacing
- Streams / Pipes? Hey, shoot me, but I like Node.js Streams/Pipes.
- Add more functions.
- Add more languages
- Node.js EEP – <https://github.com/darach/eep-js>
- React/PHP EEP - <https://github.com/ianbarber/eep-php>



- <https://github.com/darach/eep-erl>
<https://github.com/ianbarber/eep-php>
<https://github.com/darach/eep-js>



Questions?

