Handling Network Latency Variation in Modern Warfare

Mitch Sanborn

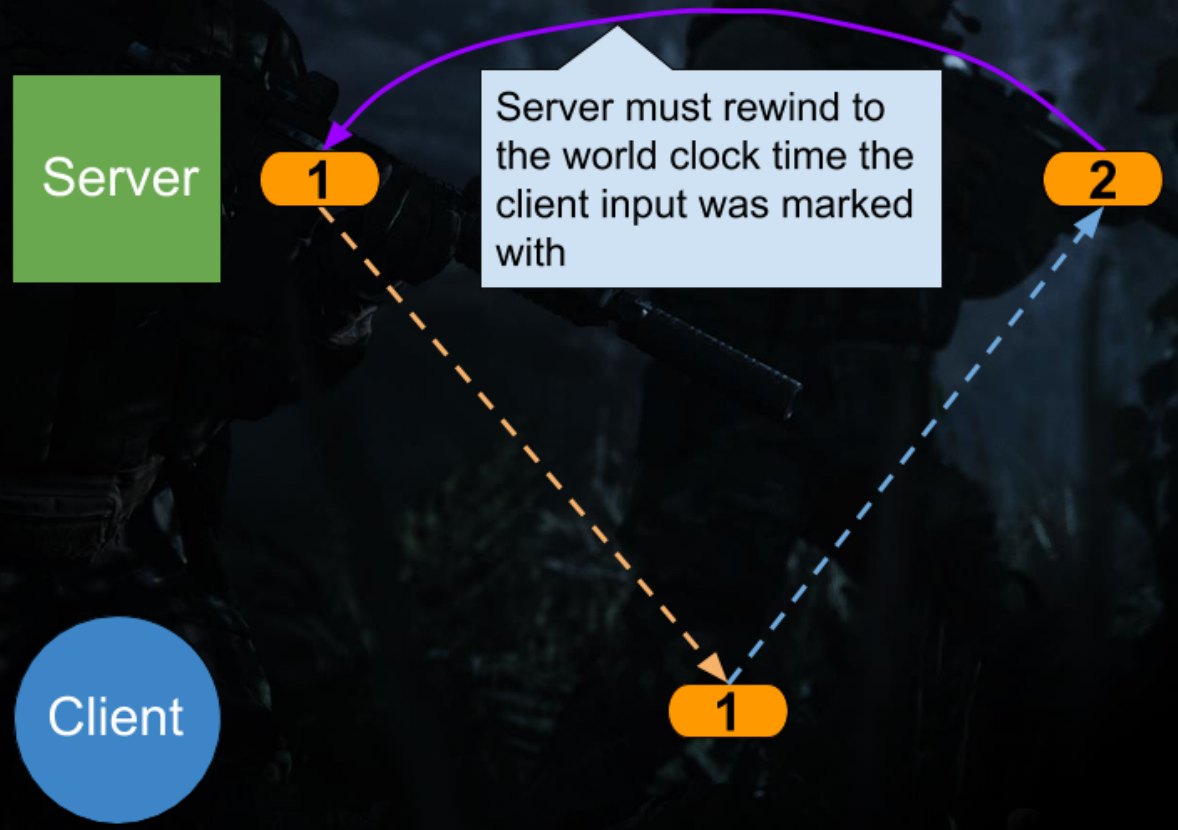
Director of Online Engineering

Infinity Ward

CALL OF DUTY MODERN WARFARE

Consequences of latent connections

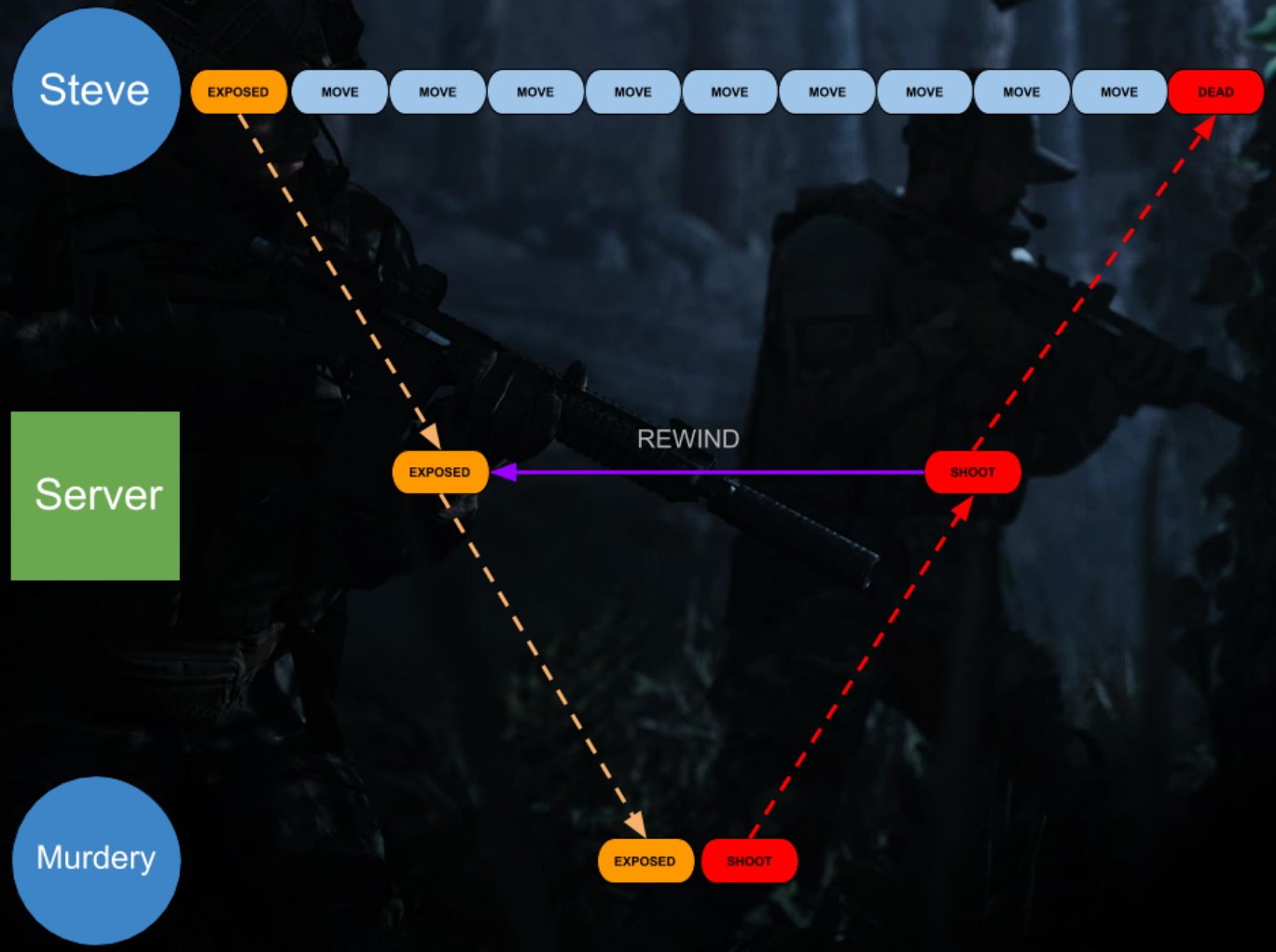
- Client input is stamped with world client time it was generated for
- Server receives input at a later world clock time
- The difference is approximately round trip latency
- For accurate bullet fire, the server rewinds time to the input time to process hit detection



Latent connectivity: A visual

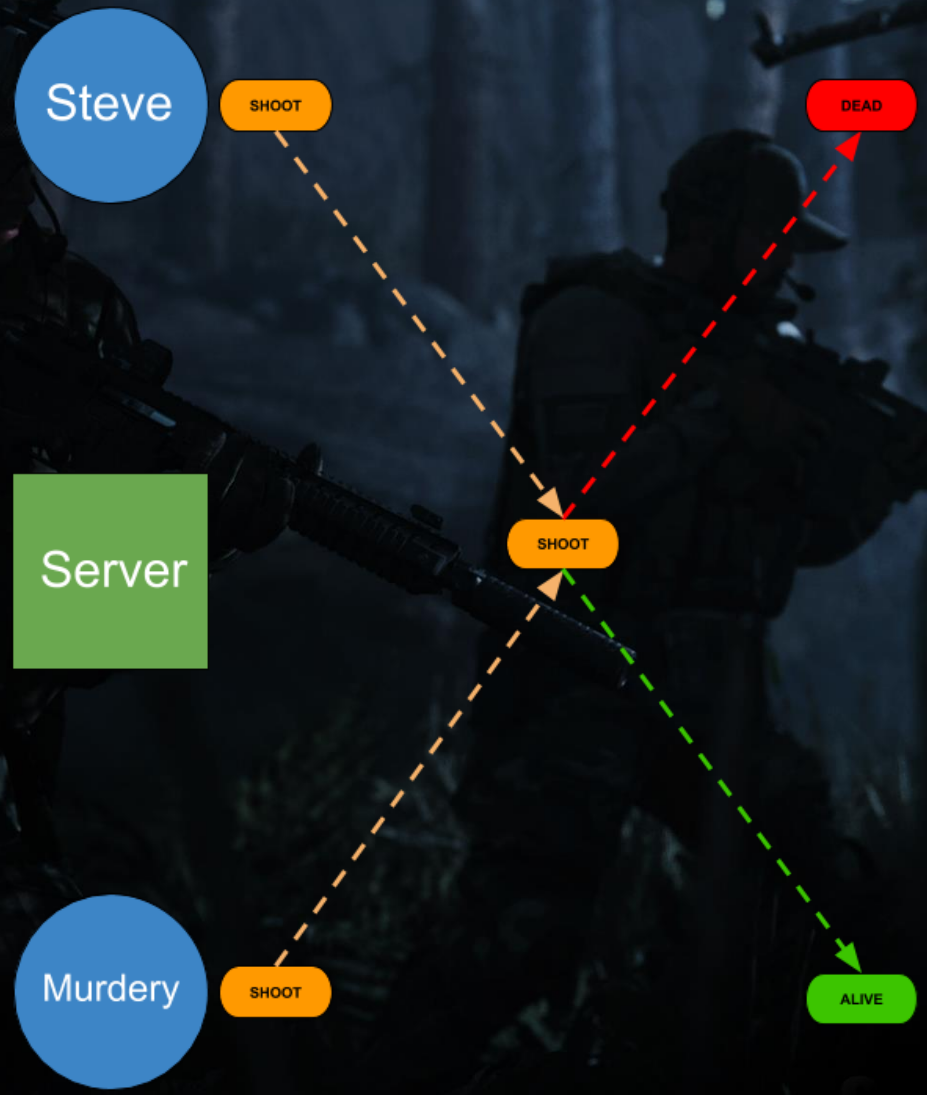
Shot behind a wall

- It is important to think about how the victim perceives antilag
- For victim, death occurs at a different location than where they were actually shot
- They may perceive themselves being shot while behind cover
- Latency exacerbates this effect



Shoot first, die first

- Latent connectivity means that players are seeing the world as it was in the past
- If two players fire their weapons simultaneously in wall clock time, the result of that weapon fire is not known until later
- Both players see themselves as having fired first but latent connectivity means the observation of the other player is delayed



Keeping latency low

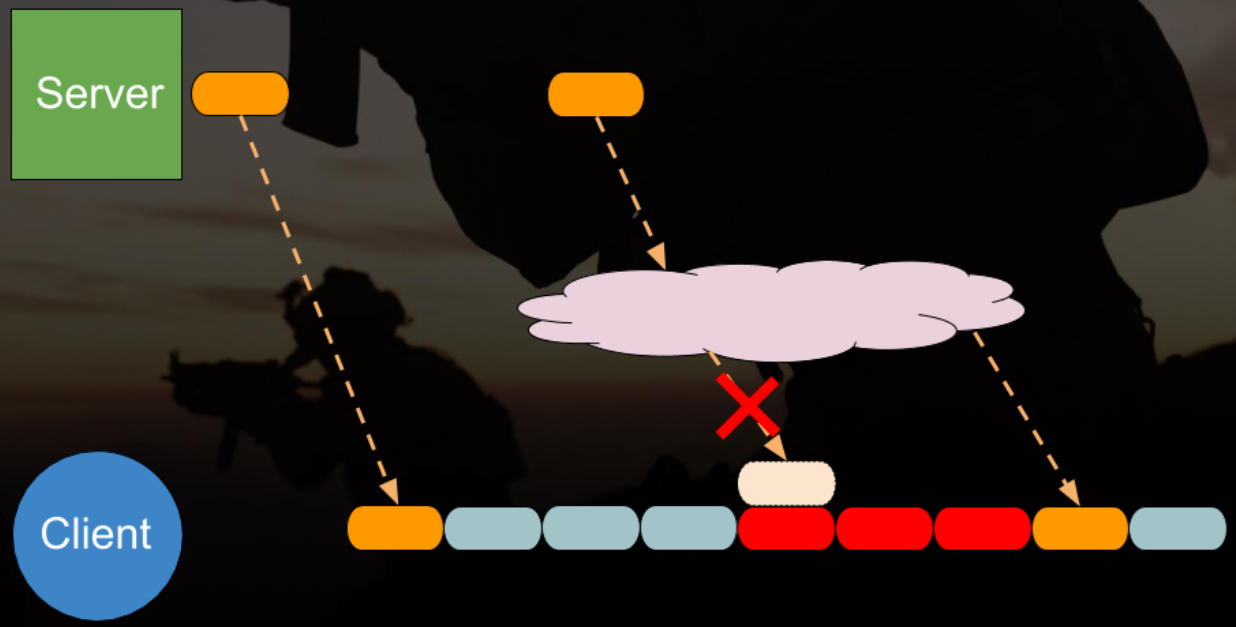
- Latency impacts gameplay in numerous ways, directly and indirectly
- This means that **keeping total system latency small has an outsized impact on the overall feel of the game**
- The systems we build to mitigate issues on the network impact overall system latency and must be built with care.

Network latency variation

- Variation in the rate of delivery of server state to the client has multiple causes
- Network related: ISP packet shaping, user router throttling, noise in the communication medium (WiFi, or collisions on a hub)
- Compute related: Server compute, Client compute
- Packet loss

Network problems break the client

- Client interpolates between server states
- Variation in packet delivery impacts the availability of server state the client can interpolate toward
- The client expects that the next server frame will be there when it needs it.
- When the client fails to find the next state when it needs it, it must guess where the objects might go (extrapolate)



Networking problems break client

Mitigating problems on the client

- Avoid running out of server data to render
- The client cannot change the rate the server delivers data
- The throttle: step the world clock faster or slower than the wall clock
- In effect, speed up or slow down the **consumption** of the server state
- Note that as we alter the rate of the world clock, we are also impacting the rate of input generation
- Apply throttle to minimize buffering but not so much that we induce error.

Network problems break the server

- The server simulates all player movement every step
- The server expects to receive approximately enough input to integrate player over the elapsed simulation time
- When the server fails to find input from a client, the server doesn't know how to move the client.

Network problems break the server

Mitigating problems on the server: Extrapolation

- Avoid running out of client input to simulate, but should data not arrive, what to do?
- The server can extrapolate client position
- Resulting position/orientation is stored in the world state and sent to clients so they perceive smooth movement
- It is not committed to the players' view of themselves.

Server Input Extrapolation

Mitigating problems on the server: Prediction

- When data doesn't arrive, just run the last command authoritatively
- Because the server is authoritative, the server can commit this position to the world state
- Position and orientation are stored both for the players own view and the world state sent to other players
- This will induce misprediction for the client that experiences the network anomaly

Server Input Prediction

Mitigating problems on the server: Buffering

- When data doesn't arrive, retain knowledge of this and attempt to commit input to a buffer
- Remember that the client's input is determined by its wall clock in conjunction with its throttle, so the client input generation is **locked** to the rate of world state generation
- So if we decide to buffer, we have to delay input instead of executing it
- This also comes with the drawback of inducing latency, as all buffering does.
- However, no misprediction or extrapolation. All input is correct.

Server Input Buffering

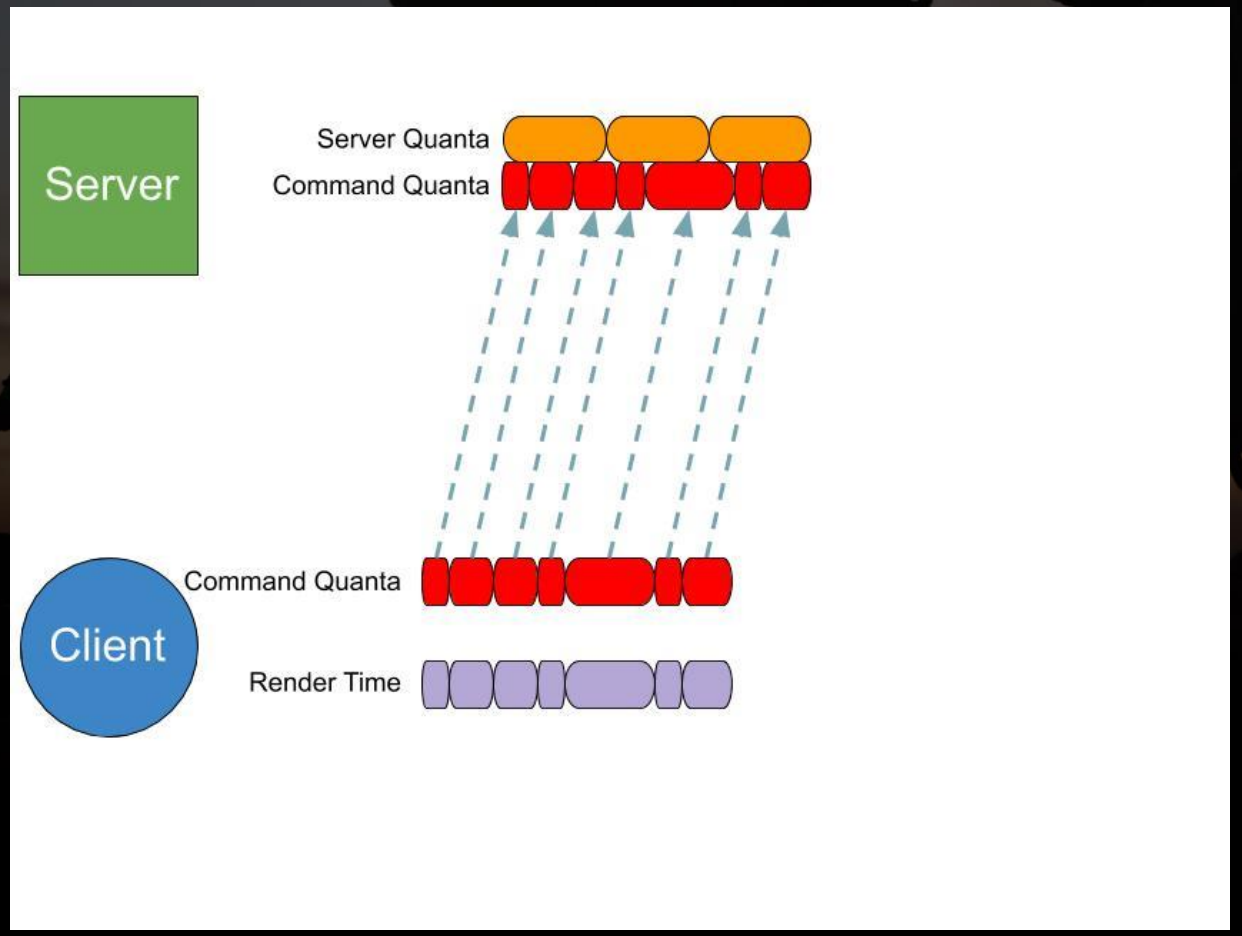
Mitigating problems on the server: *Infinite Warfare*

- Infinite Warfare had no throttle to apply to client input
- Sustained network anomalies would result in sustained extrapolation or prediction, which was not desirable
- Infinite Warfare had a 20hz server with clients targeted at 60hz, so throttling input consumption on the server was less impactful because the client would interpolate results over a coarse sample set

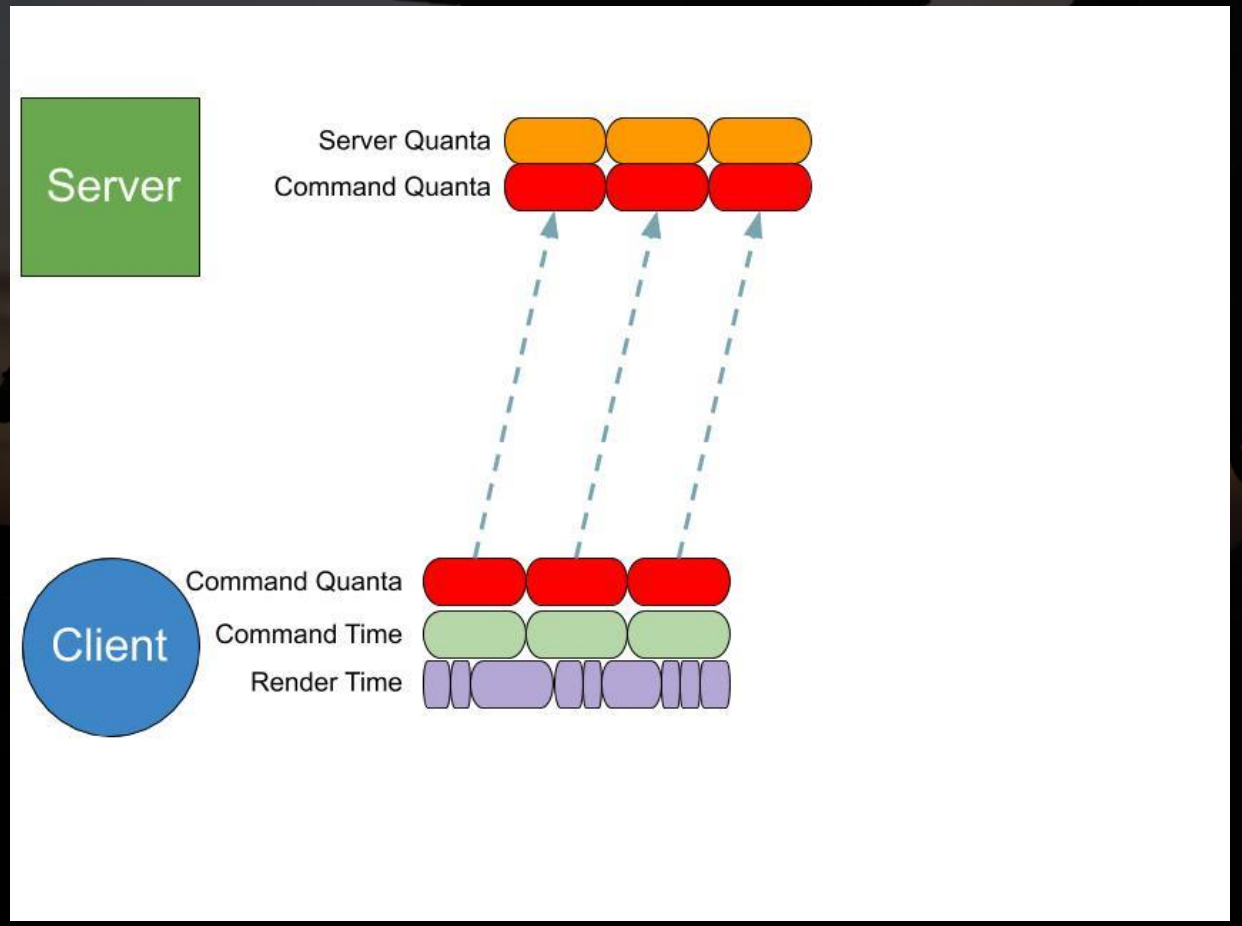
Mitigating problems on the server: *Modern Warfare*

- The lack of a throttle was troublesome. So we decided to change everything to make one.
- Instead of basing the integration over the period of **world clock** time elapsed, generate a new input clock independent of world time.
- The client integrates over the new input clock for movement and sends the clock time to the server
- A single input command quanta is fixed to some value that is less than a 60hz vsync frame (we chose 16ms)
- The client then generates a new command only when the new clock elapses one or more command quanta.

Client Input: *Infinite Warfare*



Client input: *Modern Warfare*



Creating an input throttle

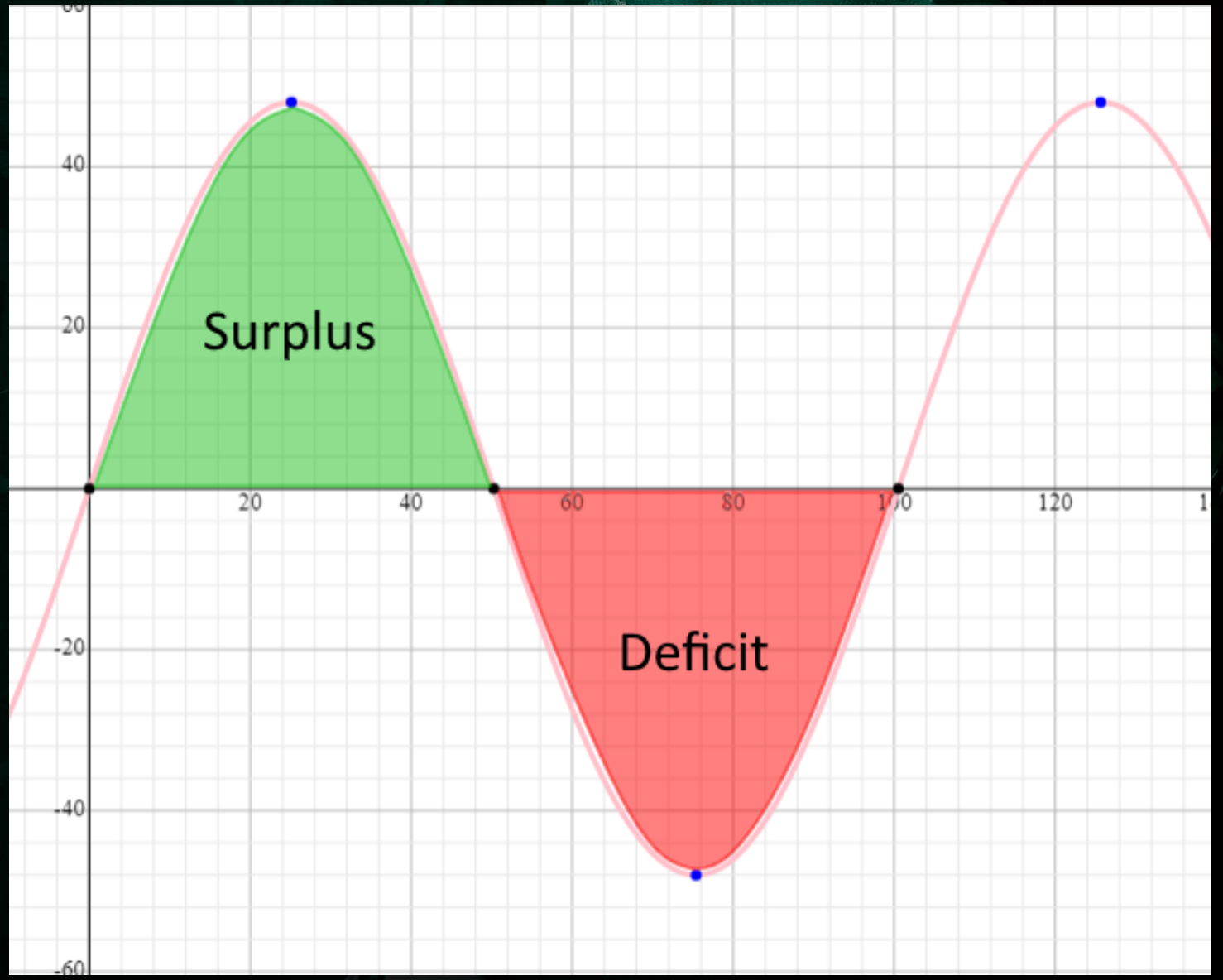
- With the new clock in place, we can now allow the server to adjust the rate of the client's command clock.
- The client sends the server the offset of the command clock from wall clock with every input.
- The server then requests more or less command data by sending the client a new target offset
- If the server is happy, the target offset is equal to the input offset
- With that change, the server can request **exactly** the amount of data it is missing

Controlling the throttle

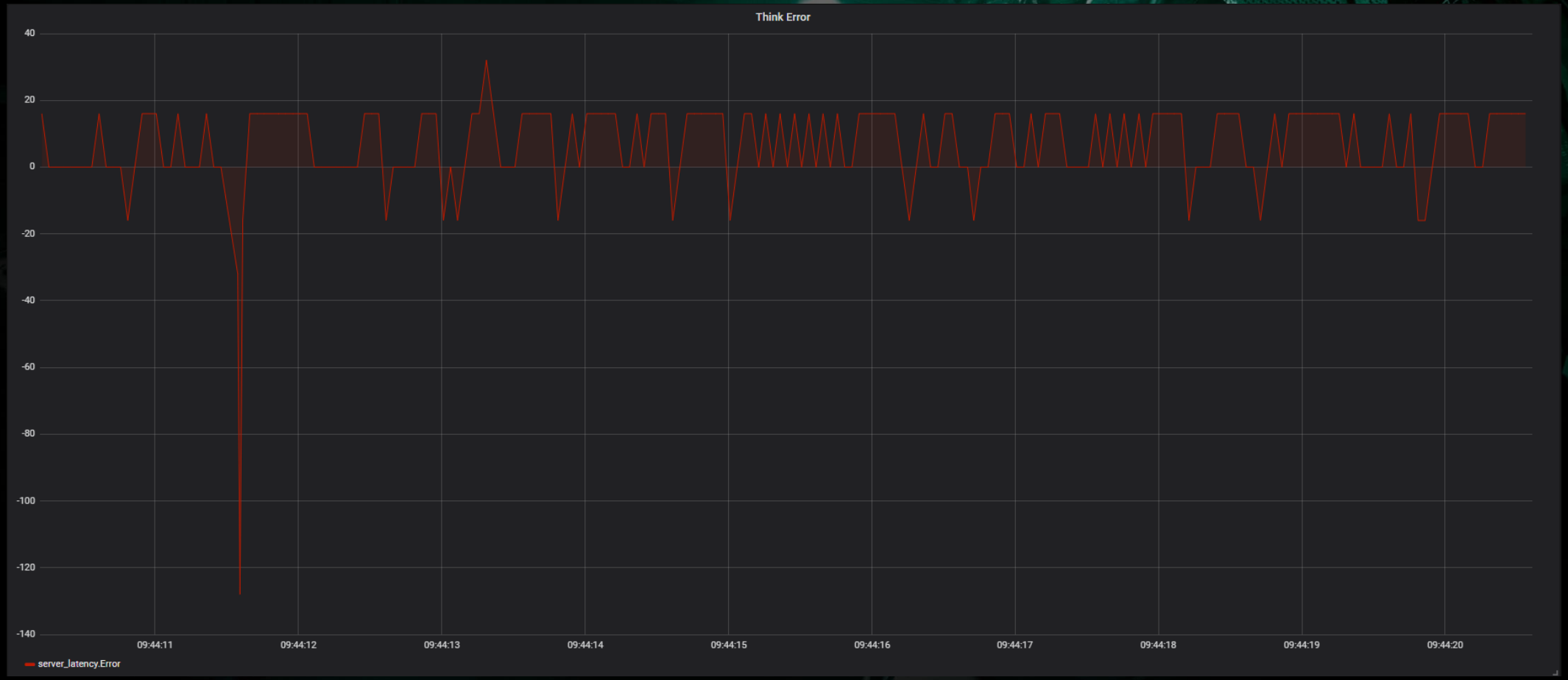
- The idea of a throttle is simple, need more = throttle up, need less = throttle down
- The unknown variable is actually "need"
- It is simple enough to say we lack data and need more, or have buffered data and need less
- This instantaneous value is quantifiable: howMuchData = bufferedData - missingData

Measuring error over time

- Difference between the amount of data buffered less the amount of data needed forms a timeseries
- Positive areas in the series represent periods of surplus data
- Negative areas in the series represent areas of deficit

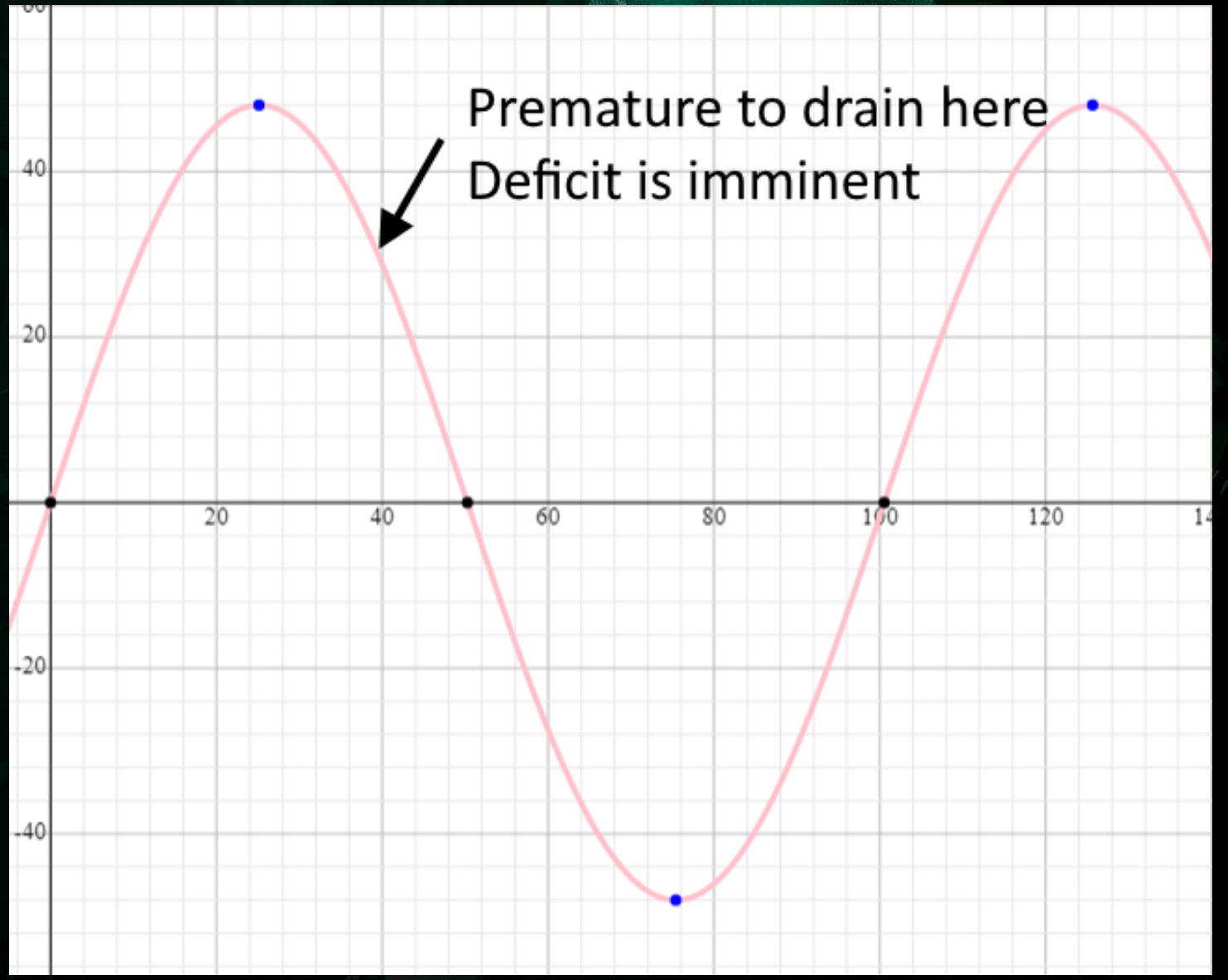


Measuring error over time

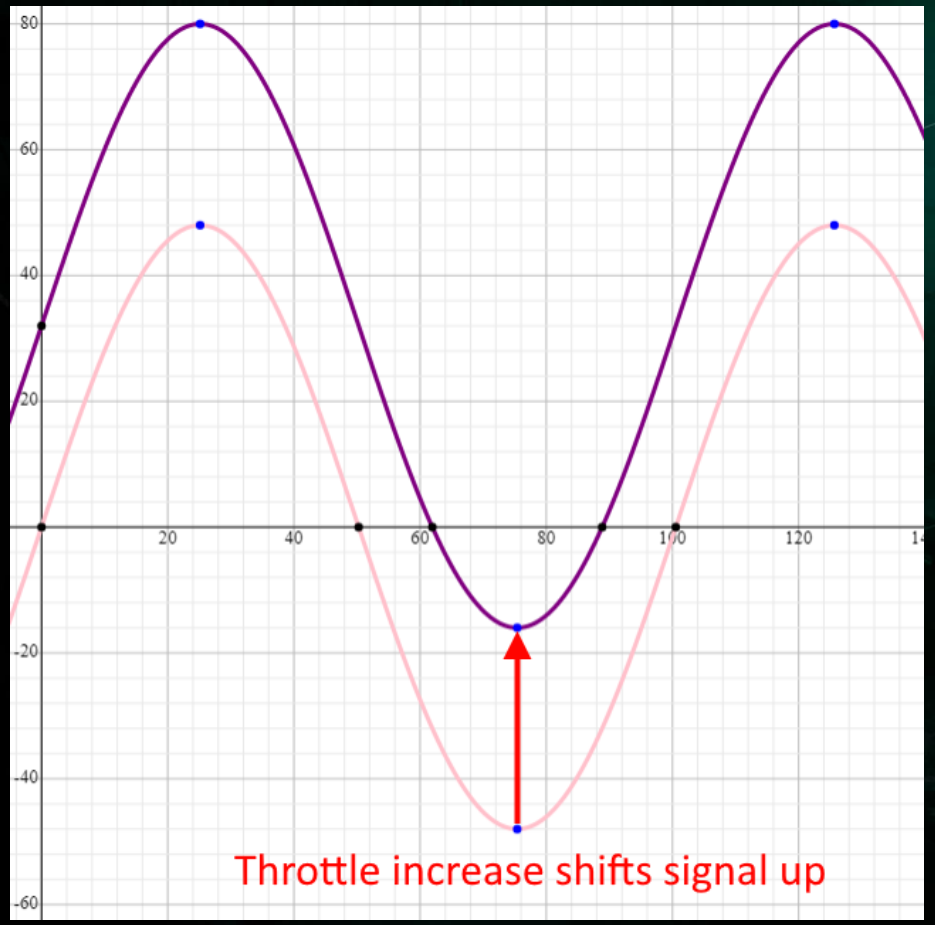


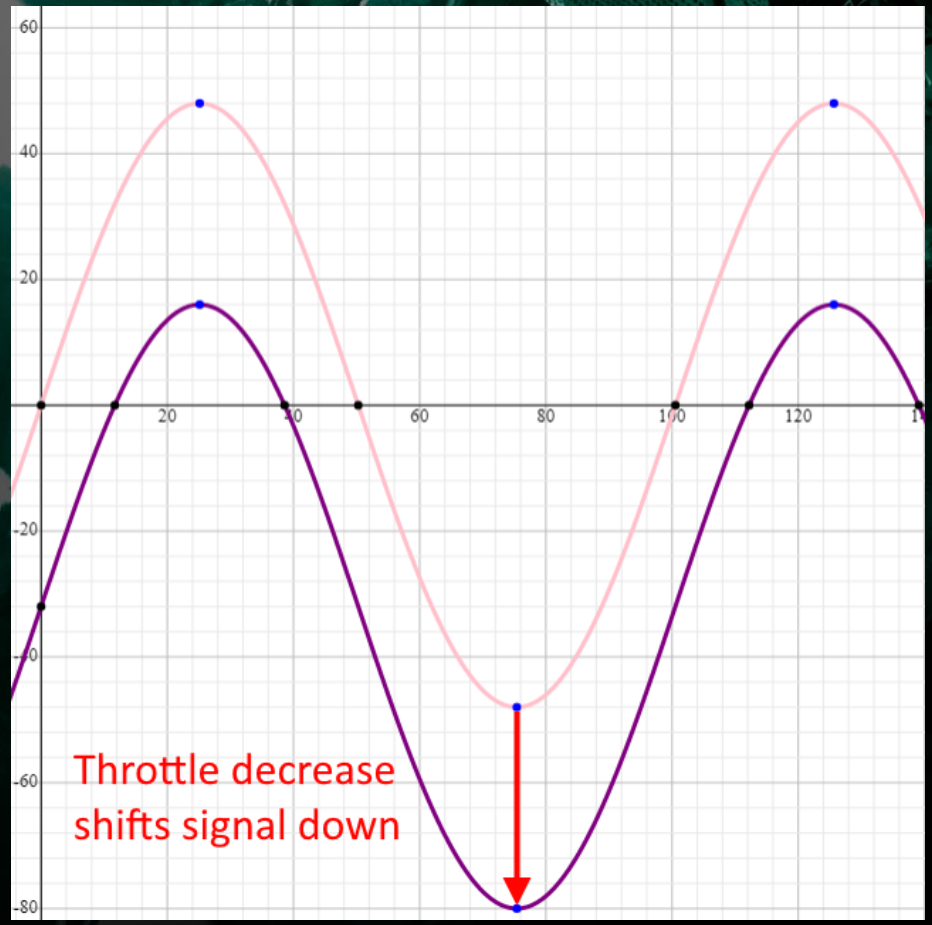
When to throttle

- A connection with sustained problems in packet delivery rate (as in noisy wifi) will have variations in the error signal over time
- In such a case, an instantaneous throttle may prematurely drain the buffer
- Ideal solution is a predictive model that can anticipate the future need for data.



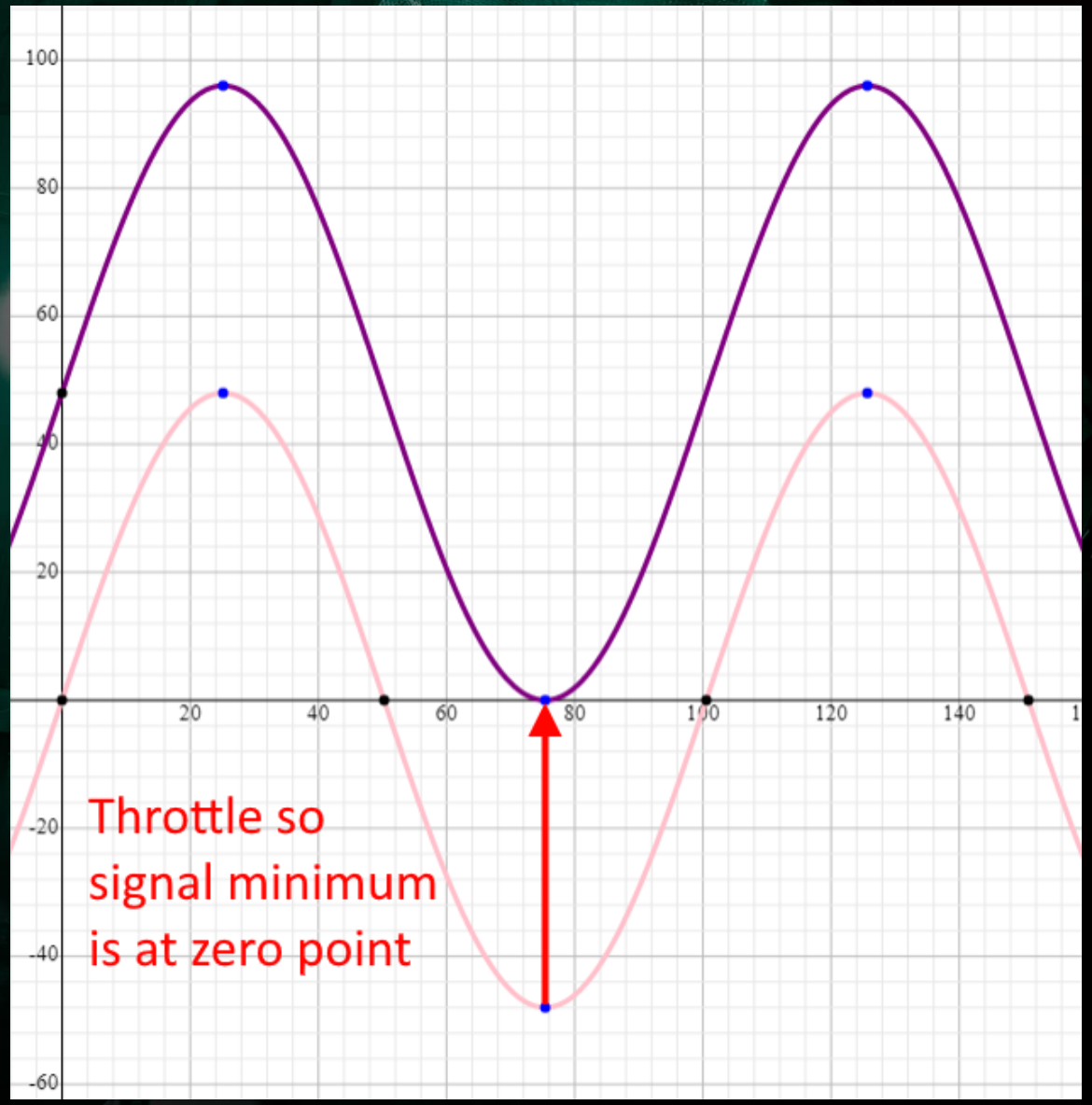
Throttling offsets the signal





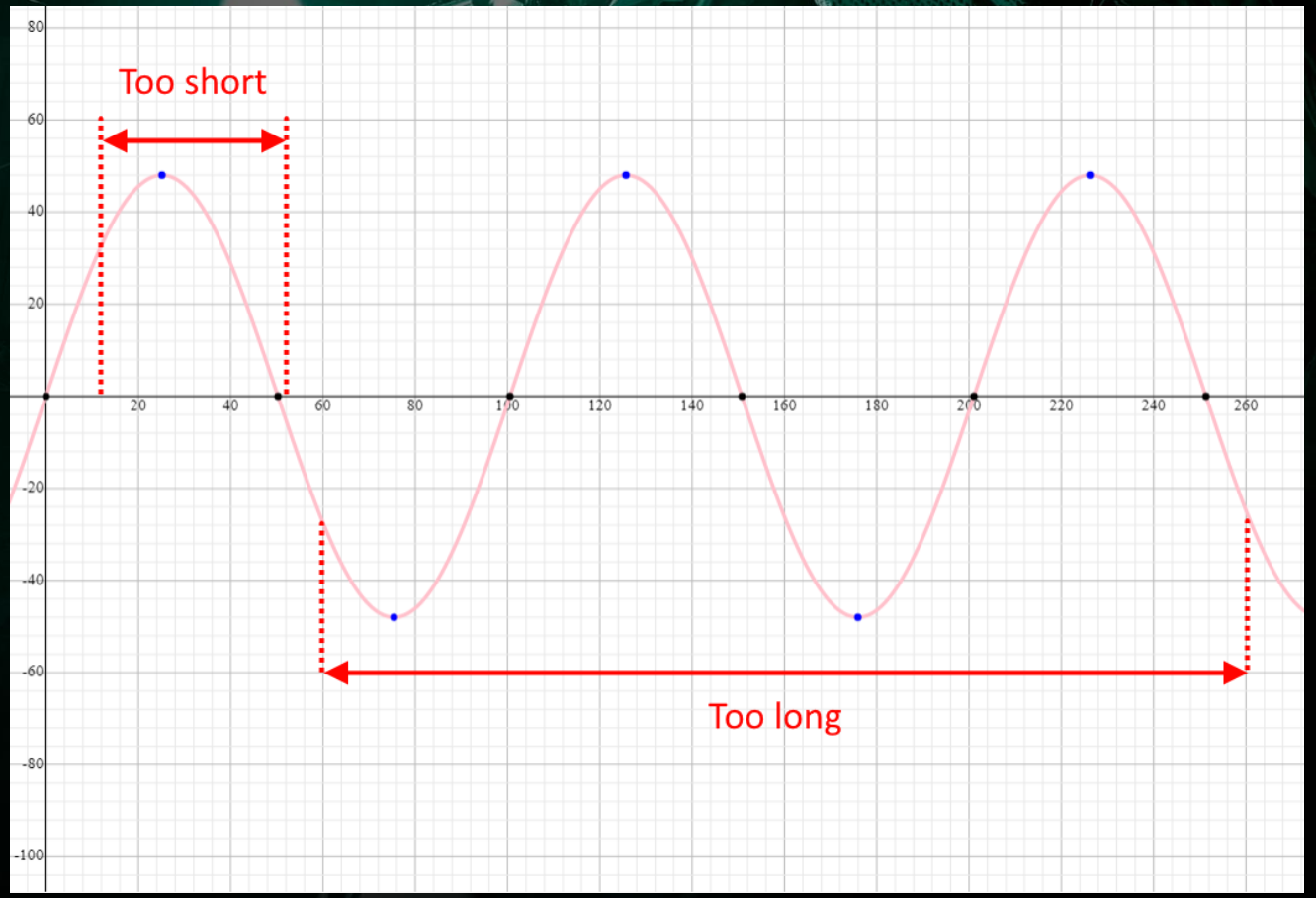
Finding the throttle magnitude

- A possible goal may be to eliminate extrapolation, preferring “correct” visualization
- For such a solution, the goal is to shift the signal up so that no deficit is incurred
- However, to minimize latency, shift no more than is required
- Shift signal so the minimum is at the zero point



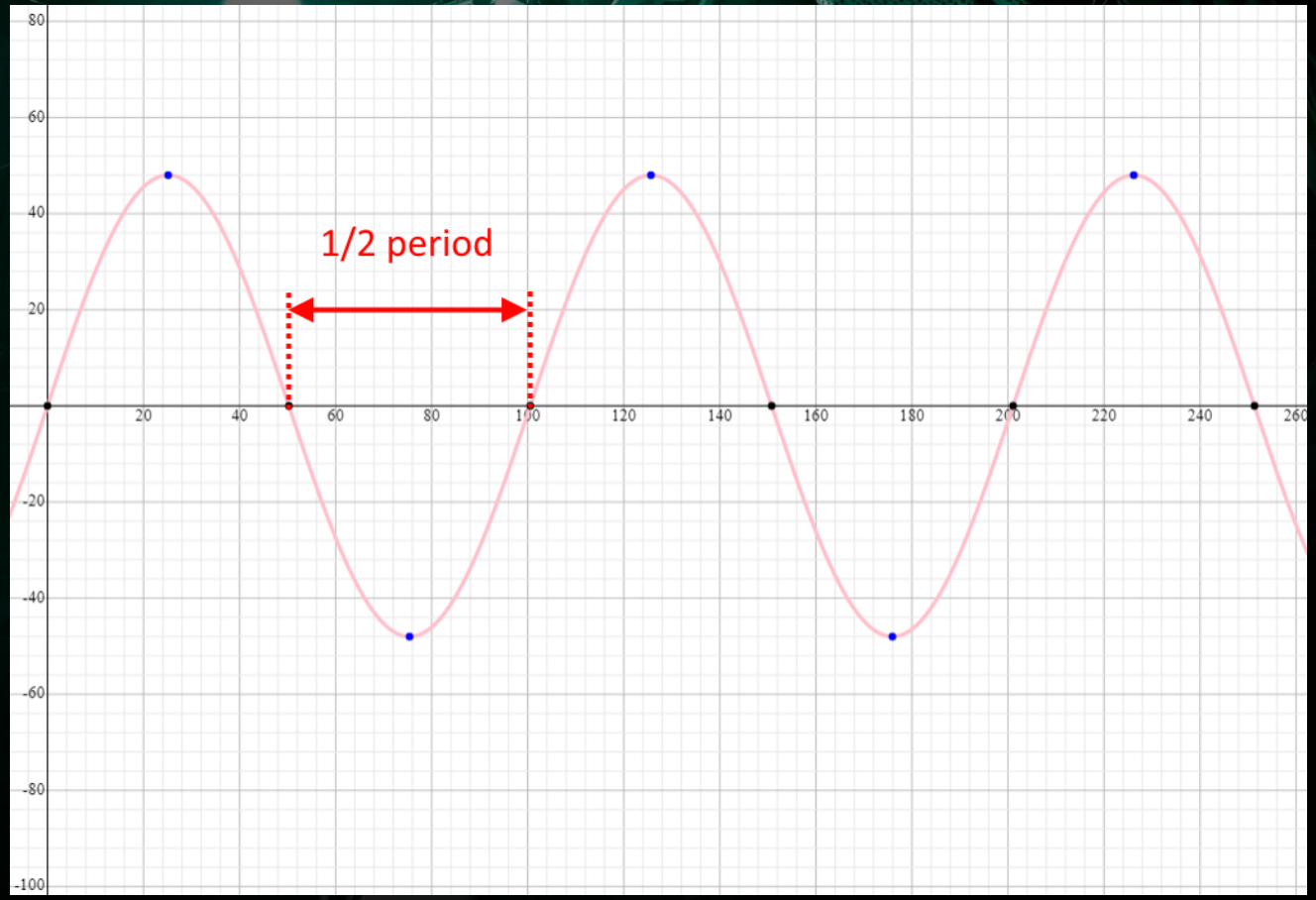
Characterizing the signal: Fixed window

- Finding the minimum of a signal is easy to see, harder to define when the signal is continuous
- One solution: Pick a reasonable fixed window size
- Too short and you fail to catch the minimum, applying the throttle incorrectly.
- Too long and you catch the minimum but the system becomes unresponsive



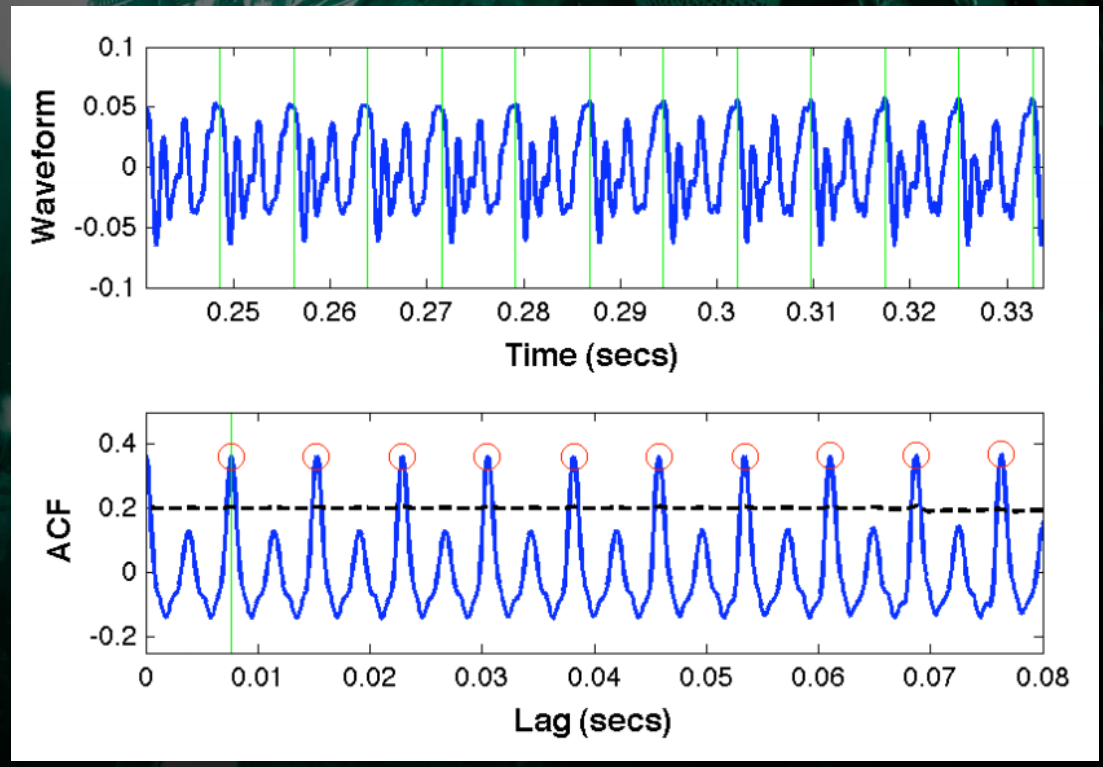
Characterizing the signal: Zero crossings

- If we can derive the periodicity of the signal, we can make the window equal to the period.
- One simple way to do this is to mark where the signal crosses the zero point
- The signal's period is double the average difference between the zero crossing points.



Characterizing the signal: Autocorrelation

- Can also use autocorrelation function.
- The autocorrelation function (ACF) at lag *k:*
  $\rho_k = \gamma_k/\gamma_0$
  γ_k = covariance(y_i, y_{i+k}) for any *i*.
  γ_0 = variance
- Covariance: $cov\,(X,Y) = \frac{1}{N}\sum_{i=1}^{N}[X_i - mean\,(X)]\,[Y_i - mean\,(Y)]$
- Variance: $var\,(X) = std\,(X)^2$



Characterizing the signal: PSD

- Power spectral density is a transform that indicates the power of the signal at different frequencies.
- The simplest way to get the power spectral density is to perform the DFT on the autocorrelation
- In this domain, we need only take the maximum of the signal. This is the frequency at which the signal is strongest
- Power spectral density is even less free that autocorrelation, adding a DFT to the mix.

Characterizing the signal: In practice

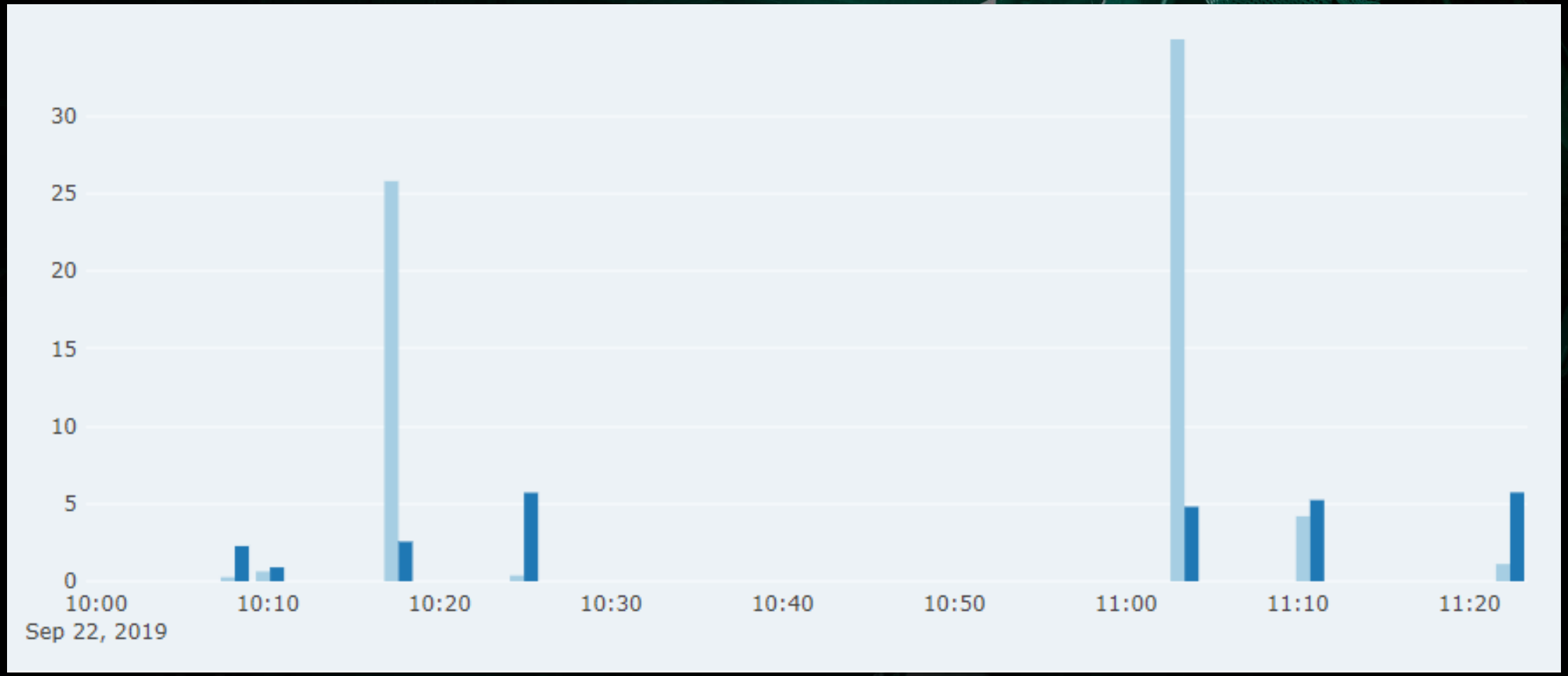
- The autocorrelation and power spectral density require a reasonable timeframe to sample from in order to derive the period
- The PSD is computationally expensive and doing this work for every client can put significant load on the server
- In the case of autocorrelation, there's a lot of handwaving that needs to be done
- Both PSD and ACF require reasonable sample size and thus retain memory longer than zero crossings, making them less responsive
- PSD and ACF are more accurate

The throttle applied

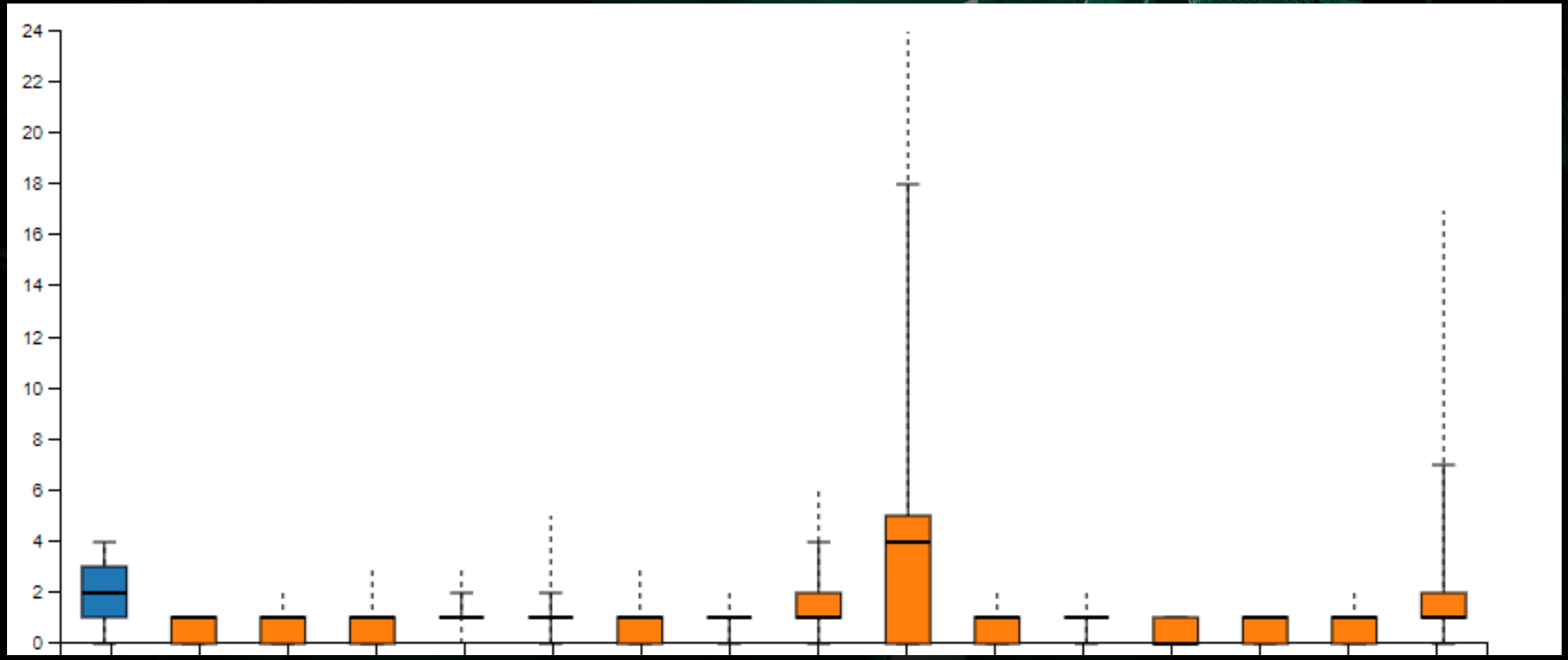
Measuring failure: The public Beta

- We had made a rather fundamental change to the network stack in adding the client input throttle and the Beta was a big test.
- One industrious user did his homework and put out a youtube video declaring the "netcode" was worse than several similar titles.
- The claim leveled was that latency was much worse
- The claim **could** be the correct if certain algorithms did not behave themselves
- Could also be the case that the algorithms were behaving themselves just fine and something was actually wrong with the connection
- The answer needed data. Which we had.

Measuring error

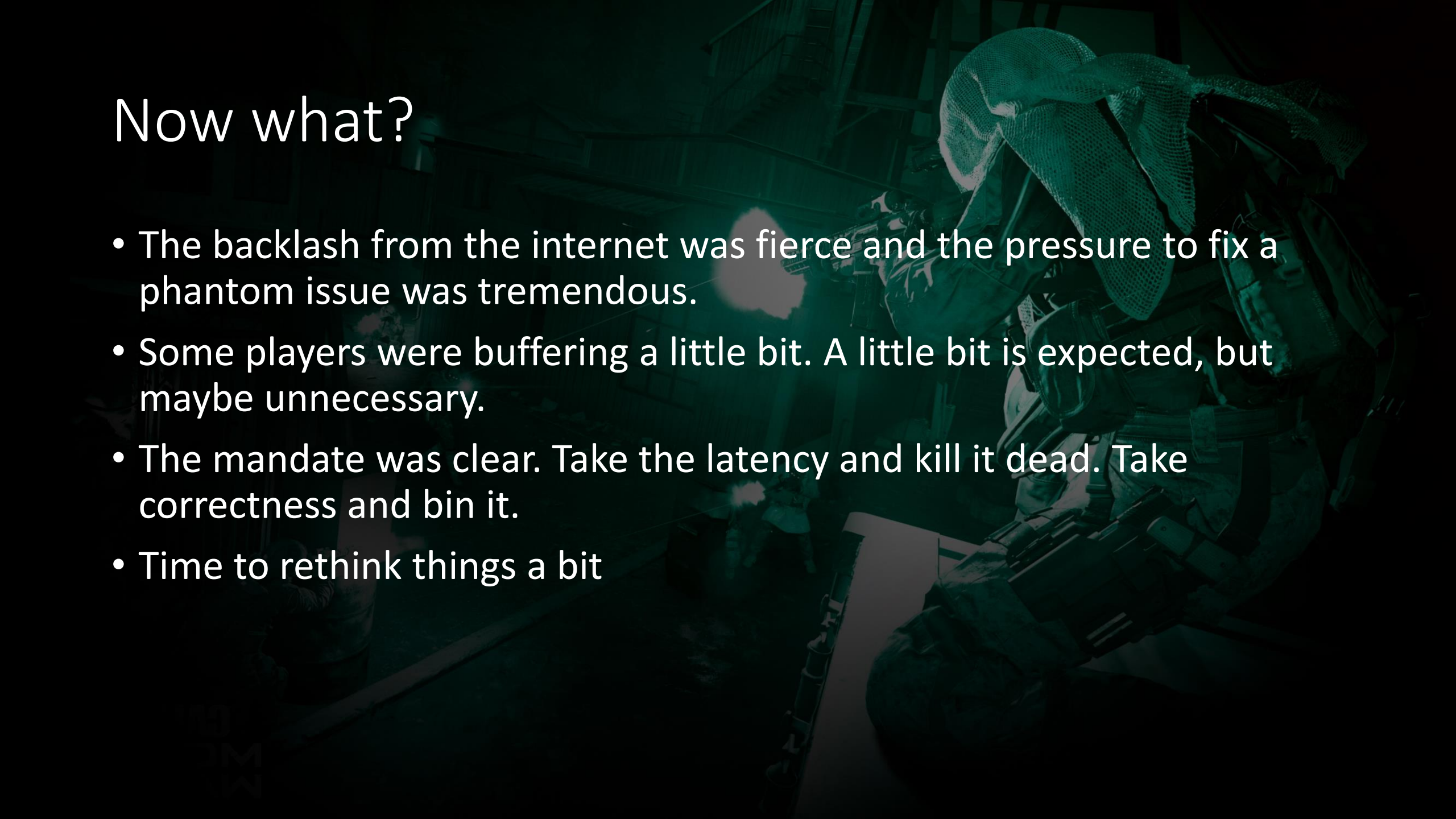


A closer look: buffering input induces latency



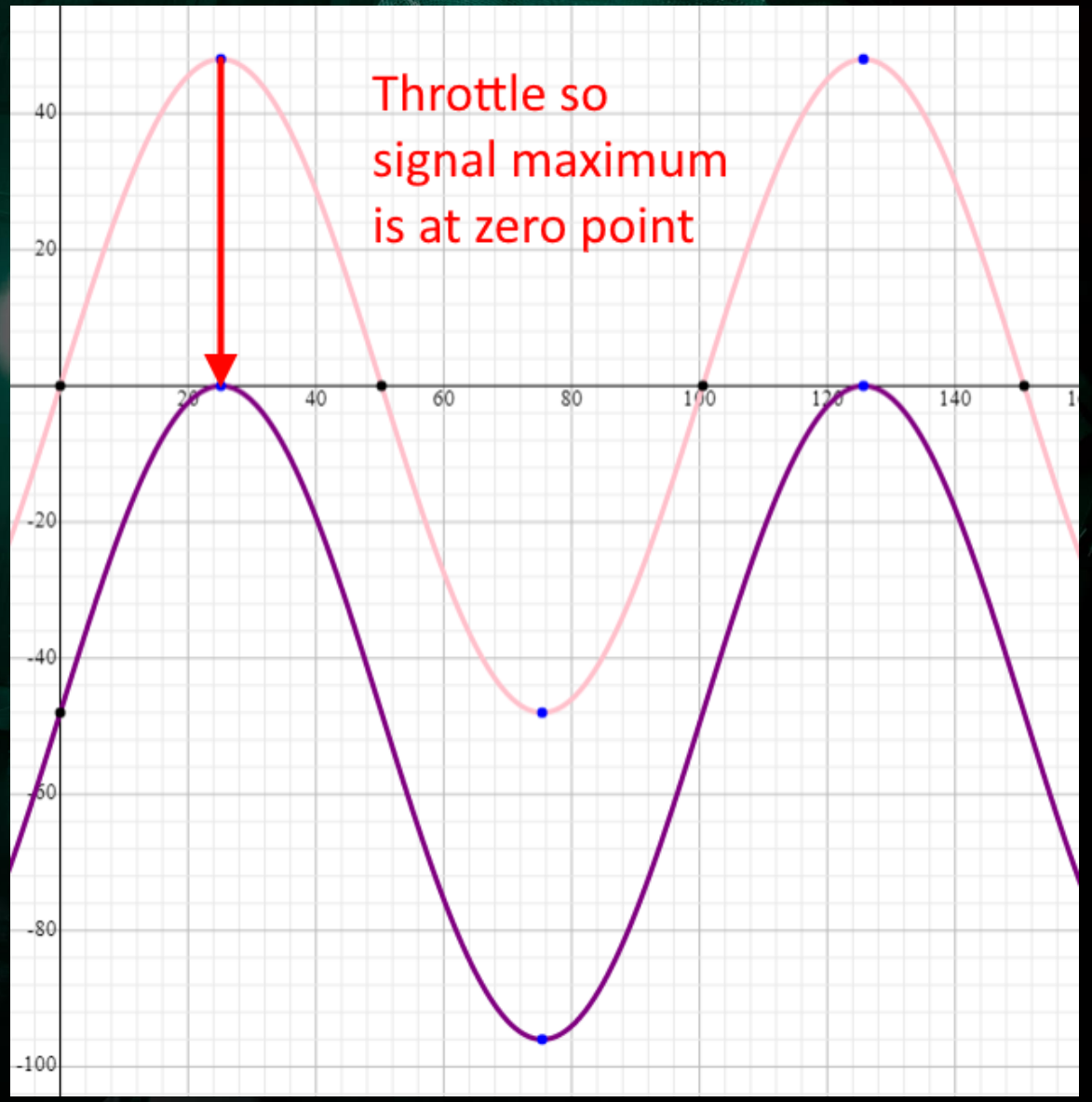
Now what?

- The backlash from the internet was fierce and the pressure to fix a phantom issue was tremendous.
- Some players were buffering a little bit. A little bit is expected, but maybe unnecessary.
- The mandate was clear. Take the latency and kill it dead. Take correctness and bin it.
- Time to rethink things a bit



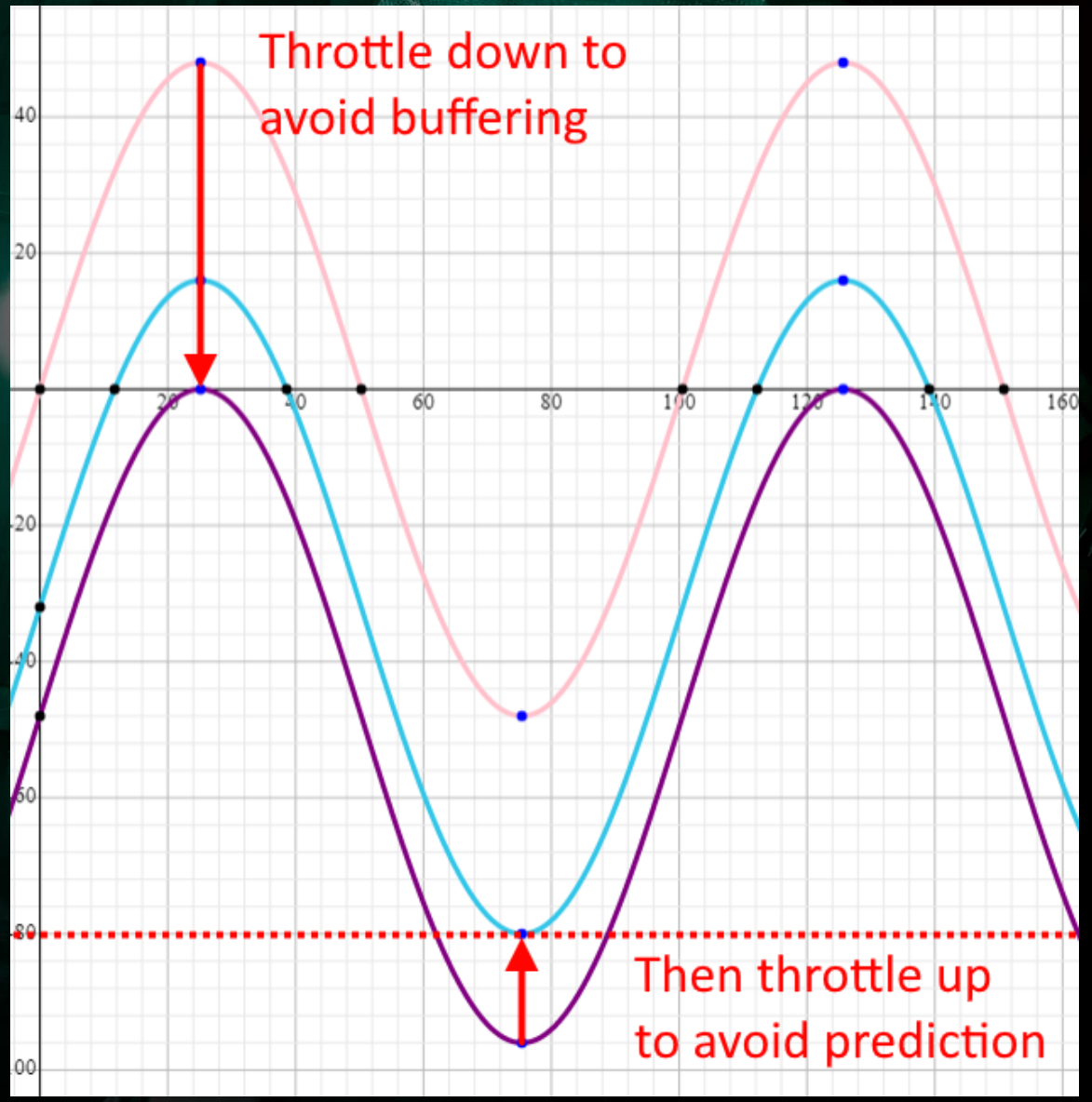
Flip flopping

- The throttle had been tuned to eliminate error at the expense of latency
- If we prefer to minimize latency we want to minimize buffering, not minimize error
- This is simple enough, just flip the problem on its head: throttle down so the maximum surplus is at the zero point

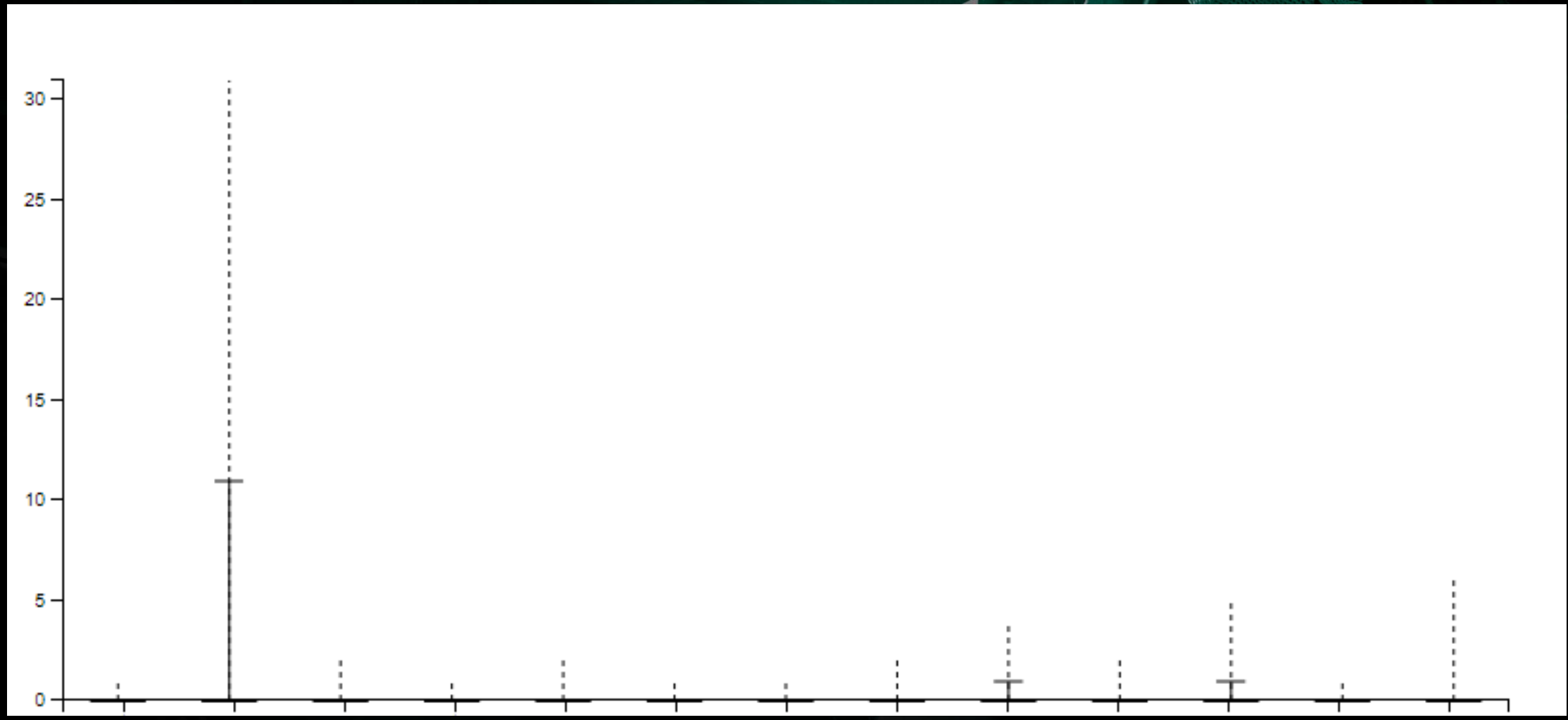


Dialing it back

- Extrapolation is not awesome and starts to break down at some point
- Prediction is also not awesome as it induces client problems
- Define limit where extrapolation stops and prediction begins
- Throttle down to avoid buffering for the fast majority of players, but throttle up to avoid prediction



Reworking throttle to eliminate latency

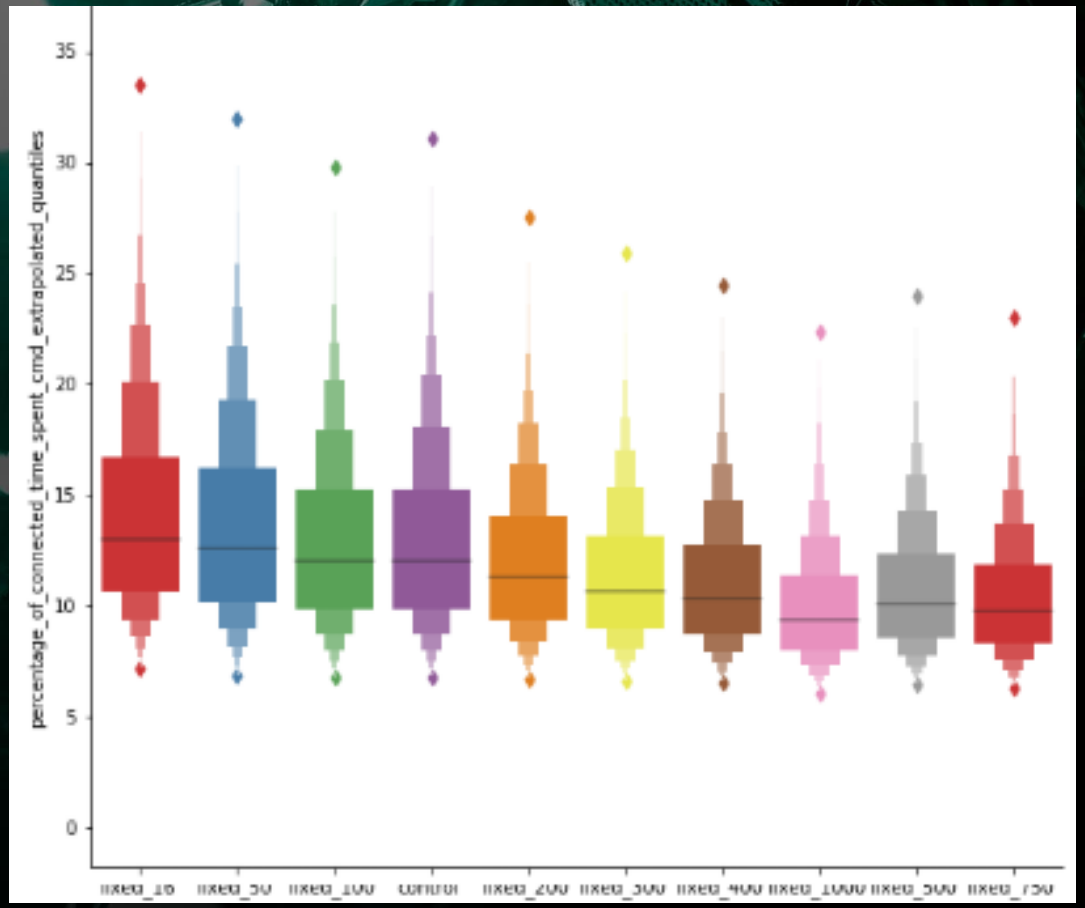


More to do: Zero crossings

- What about the periodicity. Did we get it right?
- Detecting signal period is predicated on the signal being periodic. Is it?
- Is there some window that would serve us better than what we dynamically generate?
- Hypothesis: Generally speaking a larger window size results in less extrapolation at the expense of greater latency
- We developed an experimental framework to test a multitude of hypothesis in the live game. This seemed like a good way to stretch its legs.

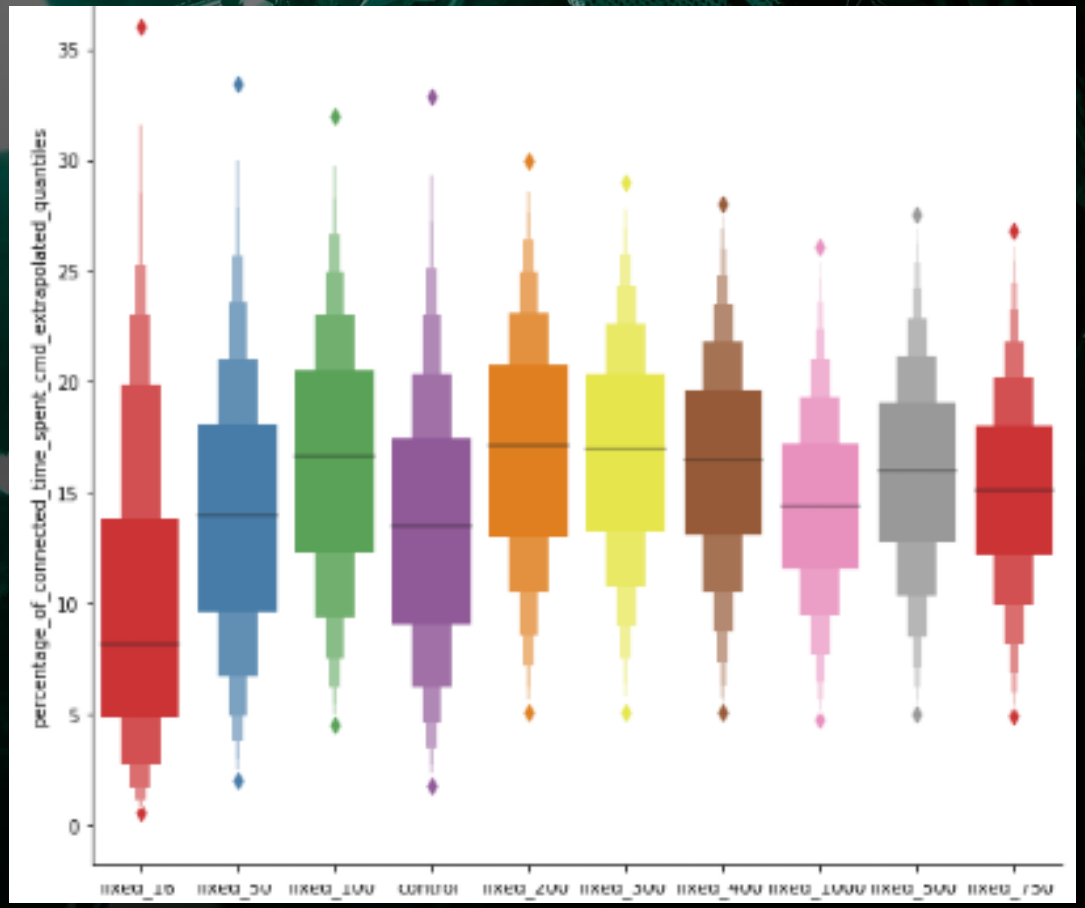
What we found: console extrapolation

- For console players, our hypothesis is largely correct.
- The dynamic window is the purple control of the experiment
- The dynamic window ends up being... not the best or the worst.
- It isn't a bad compromise. But it's less than ideal.



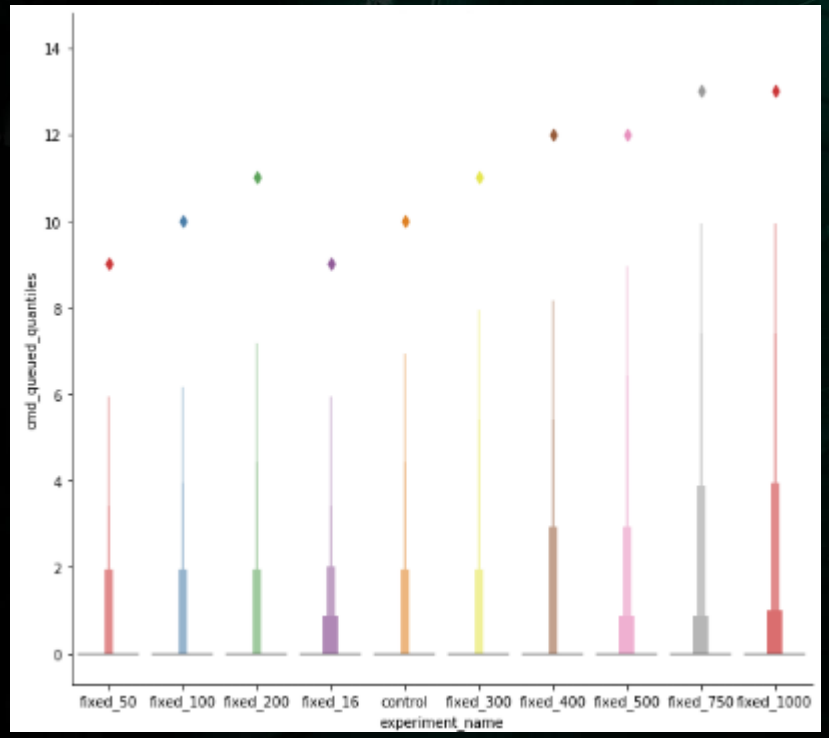
What we found: PC extrapolation

- For PC, the story is different
- For PC players, a 16ms window, an instantaneous response, is clearly superior which is contrary to the consoles.
- The curve is also non-linear
- The dynamic window performs admirably here, but is still not as good as a 16ms instantaneous window.

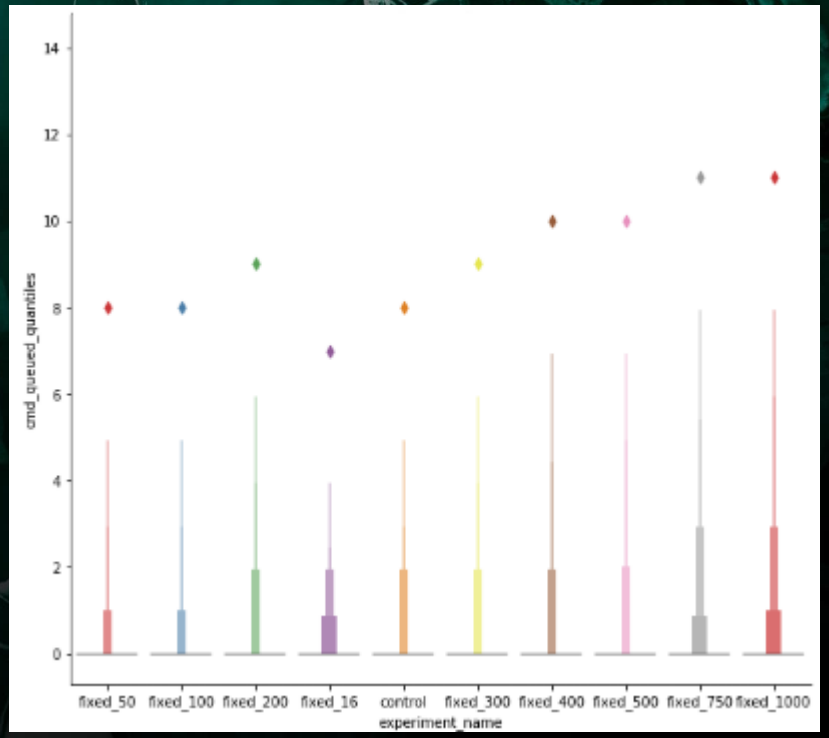


What we found: Buffering

Console Buffering

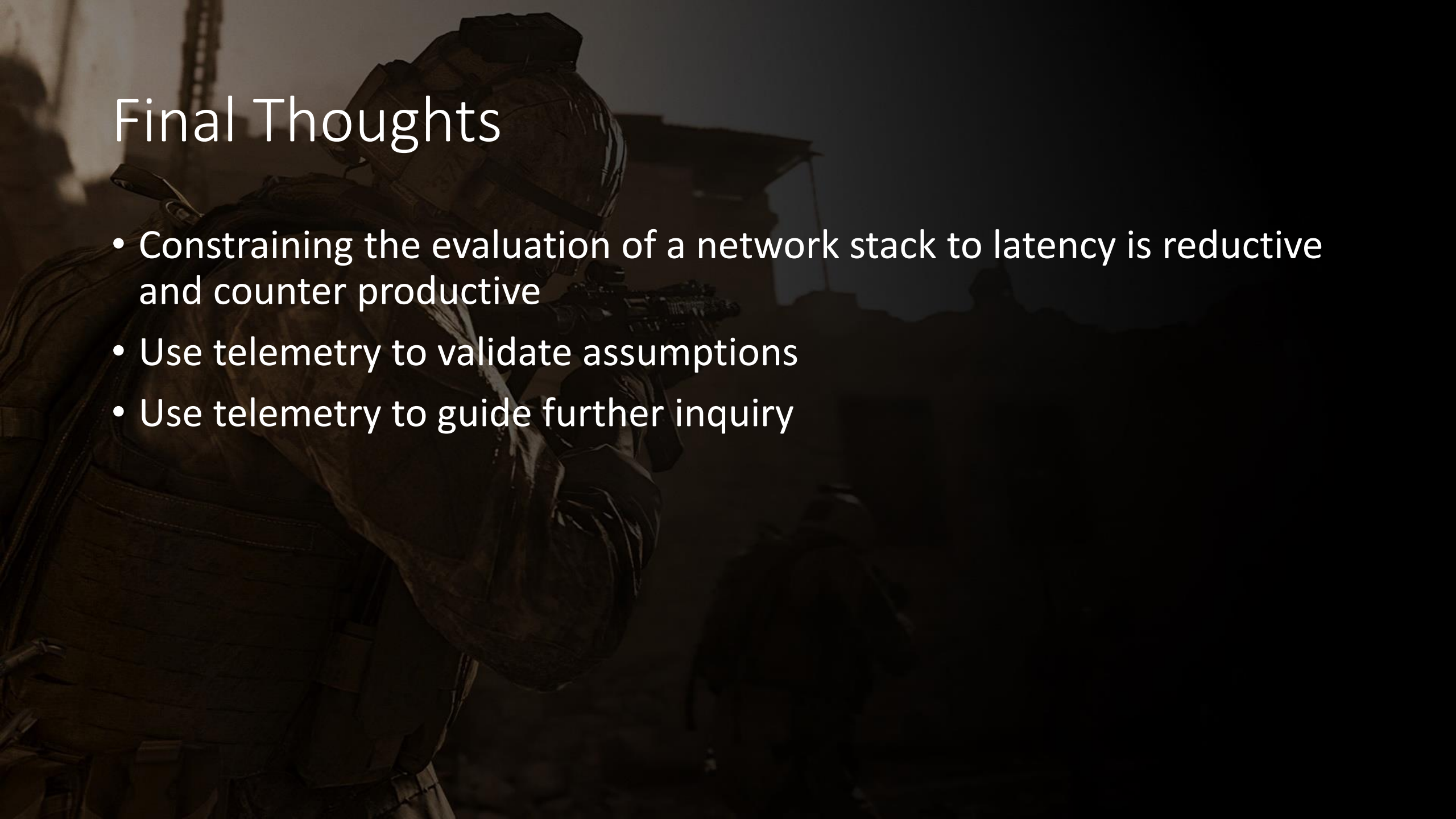


PC Buffering



Moving forward

- Dynamic window is kinda ok, certainly a reasonable compromise for consoles.
- PC is its own beast and work remains there to understand the problem space better, but the data clearly indicates an instantaneous throttle is superior.
- Research into the kinds of signals being observed is maybe warranted.
- The overall system latency here is pretty low given the architecture we have. Without dramatic architectural changes, improving it will be incremental at best.



Final Thoughts

- Constraining the evaluation of a network stack to latency is reductive and counter productive
- Use telemetry to validate assumptions
- Use telemetry to guide further inquiry

Special thanks

- John Dennison
  - Data Guru
- Geoff Evans
  - He made me do it
- IW Gameplay team
  - I made so… many… bugs…

CALL OF DUTY MODERN WARFARE