DynFilter: Limiting Bandwidth of Online Games using Adaptive Pub/Sub Message Filtering

Julien Gascon-Samson, Jörg Kienzle, Bettina Kemme

School of Computer Science, McGill University
Montreal, Canada

December 3th, 2015

Introduction and Background

1 Introduction and Background

2 DynFilter Architecture

3 Load Analysis & Optimization

4 Experiments

5 Conclusion

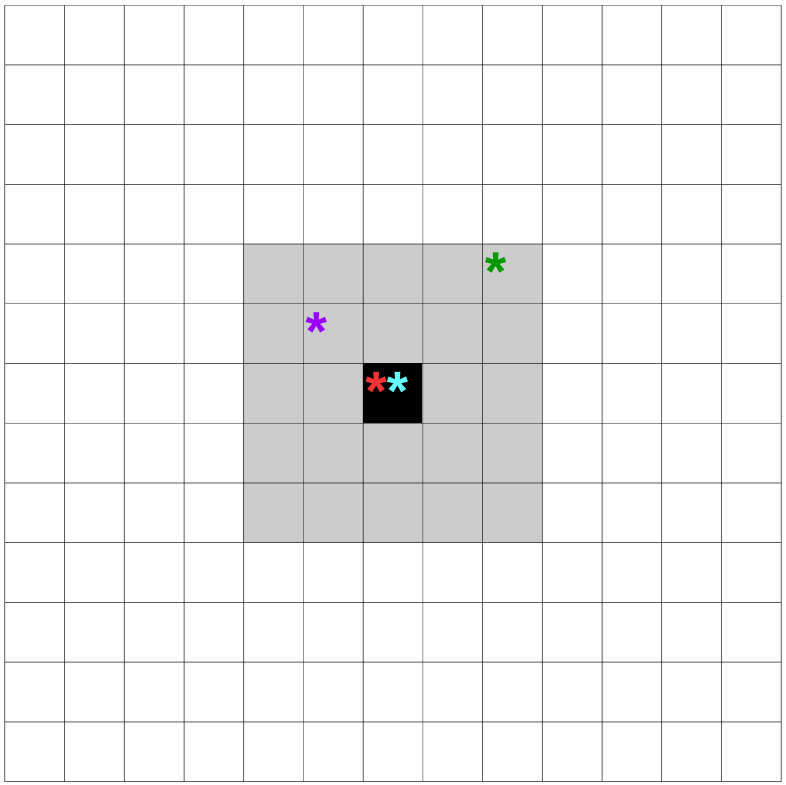
DynFilter Overview - Managing Bandwidth

- Managing bandwidth: often a concern in multiplayer games

DynFilter Overview - Managing Bandwidth

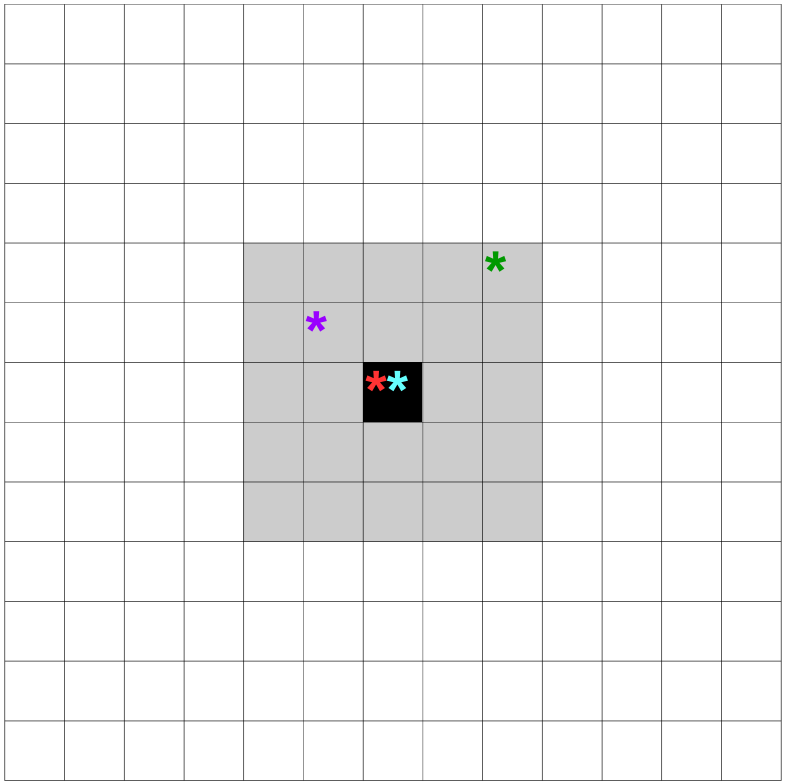
- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

DynFilter Overview - Managing Bandwidth



- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

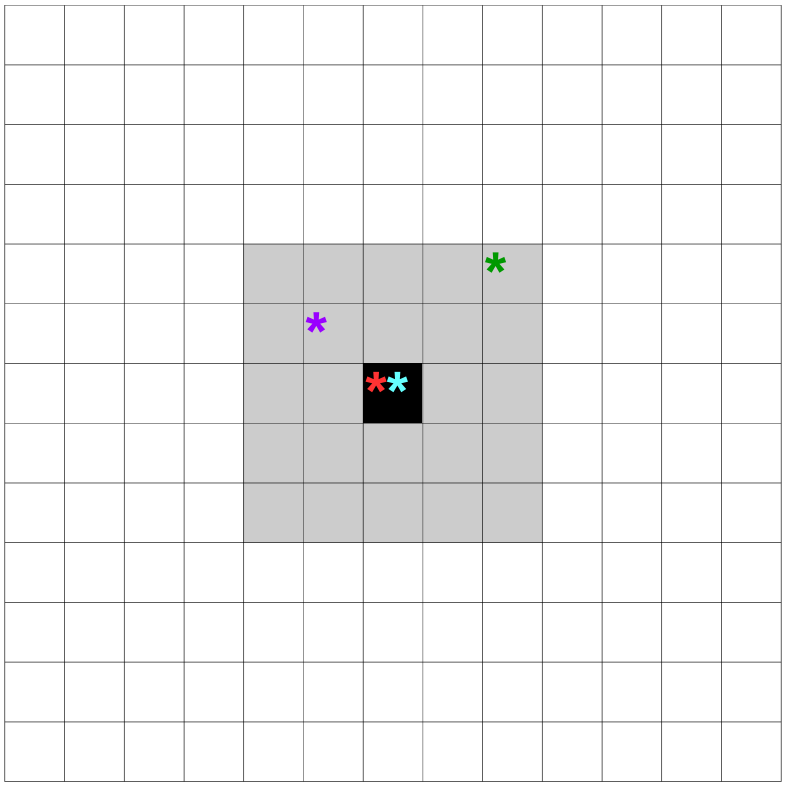
DynFilter Overview - Managing Bandwidth



- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

- Idea: game operator defines a target bandwidth "quota" over a window

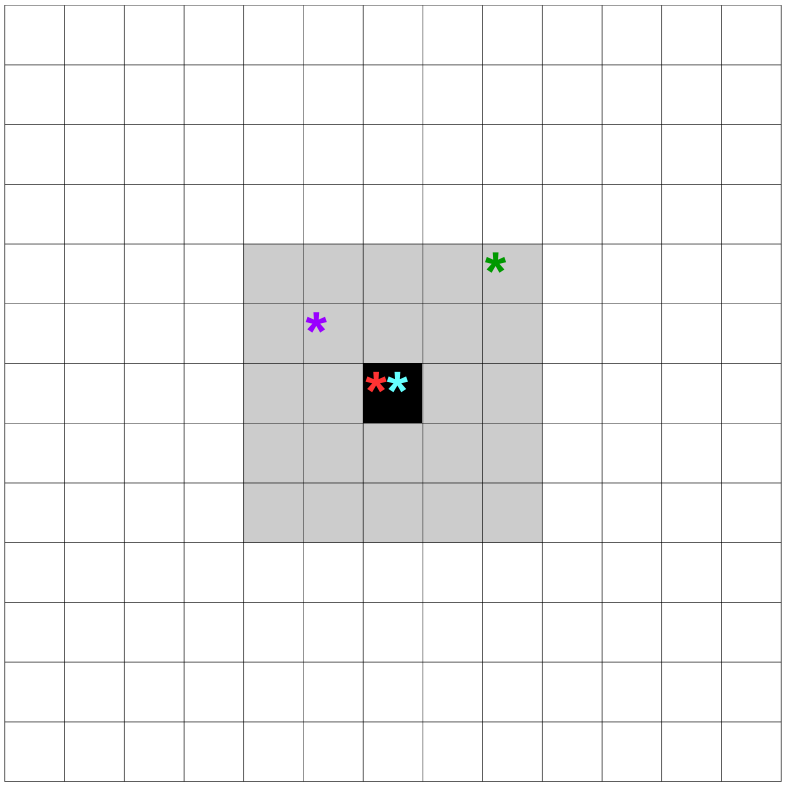
DynFilter Overview - Managing Bandwidth



- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

- Idea: game operator defines a target bandwidth "quota" over a window
- DynFilter will apply filtering in order to meet the predefined quota

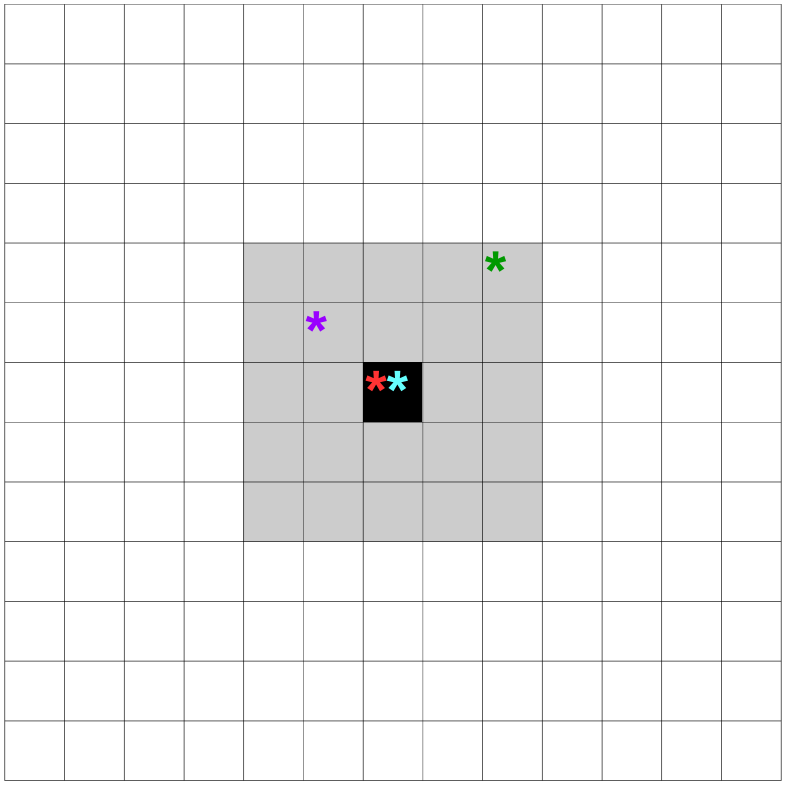
DynFilter Overview - Managing Bandwidth



- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

- Idea: game operator defines a target bandwidth "quota" over a window
- DynFilter will apply filtering in order to meet the predefined quota
- Continuously readjusted

DynFilter Overview - Managing Bandwidth



- Managing bandwidth: often a concern in multiplayer games
- **DynFilter: reducing the rate at which state updates are sent for "remote" entities**

- Idea: game operator defines a target bandwidth "quota" over a window
- DynFilter will apply filtering in order to meet the predefined quota
- Continuously readjusted
- Offered as a Cloud-based platform

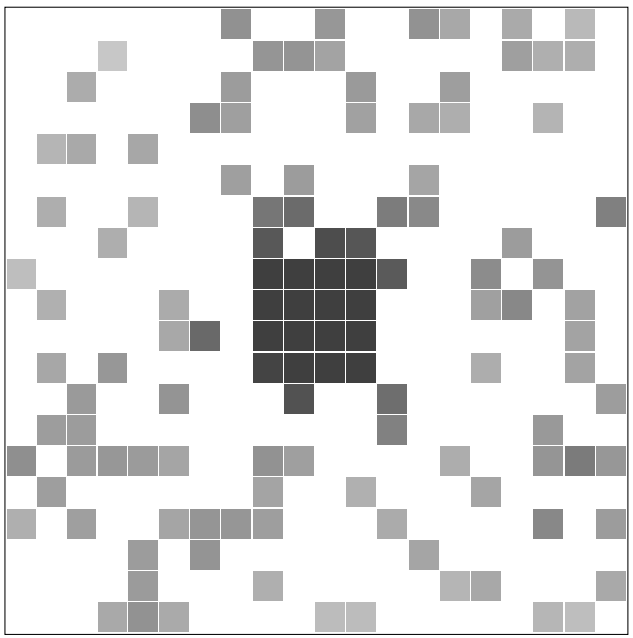
DynFilter Overview - Tile-Specific Filtering

- Players not evenly distributed in multiplayer games (ie. flocking)

DynFilter Overview - Tile-Specific Filtering

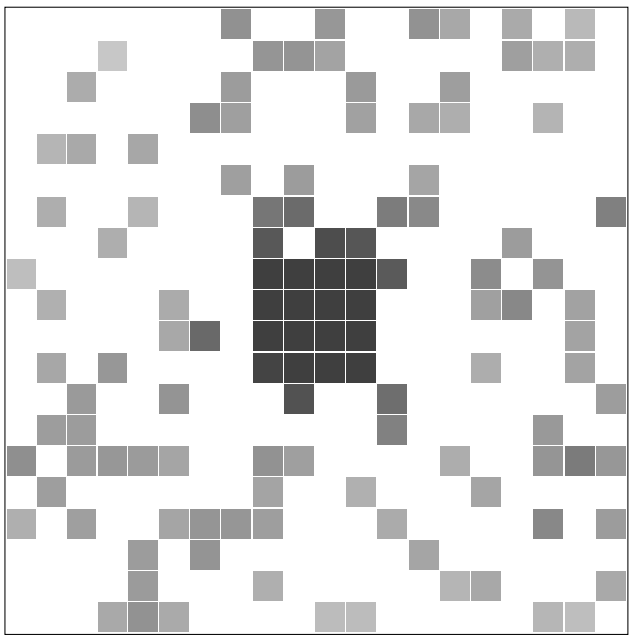
- Players not evenly distributed in multiplayer games (ie. flocking)
- Idea: varying amount of filtering based on tile "density"

DynFilter Overview - Tile-Specific Filtering



- Players not evenly distributed in multiplayer games (ie. flocking)
- Idea: varying amount of filtering based on tile "density"

DynFilter Overview - Tile-Specific Filtering



- Players not evenly distributed in multiplayer games (ie. flocking)
- Idea: varying amount of filtering based on tile "density"

- More players:
  - State updates of each individual player less important
  - Larger bandwidth usage

Maintaining Immersion in Games

Game state updates must be delivered within certain time bounds and/or at specific frequencies

Maintaining Immersion in Games

Game state updates must be delivered within certain time bounds and/or at specific frequencies
FPS games:

- Very fast-paced
- Quake: ~ 20 updates per second (50 ms between updates)

Maintaining Immersion in Games

Game state updates must be delivered within certain time bounds and/or at specific frequencies

FPS games:

- Very fast-paced
- Quake: ~ 20 updates per second (50 ms between updates)

RPG / MMORPG games:

- Slower-paced
- A few updates per second (highly dependant on the game)

Maintaining Immersion in Games

Game state updates must be delivered within certain time bounds and/or at specific frequencies

FPS games:

- Very fast-paced
- Quake: ~ 20 updates per second (50 ms between updates)

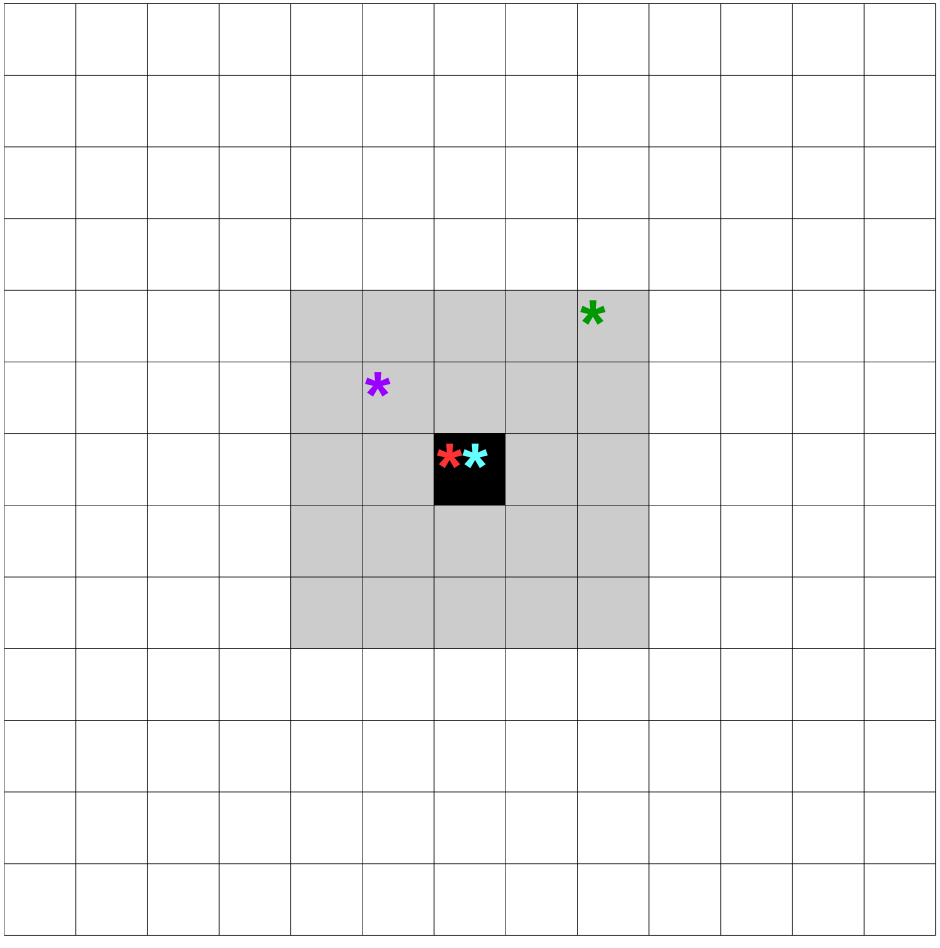
RPG / MMORPG games:

- Slower-paced
- A few updates per second (highly dependant on the game)

If state updates are too infrequent:

- Perception of "lag"
- Players / objects jumping
- Reduction in fun factor

Interest Management



- Goal: limiting the amount of messages that need to be transmitted
- No need to transmit state updates from all entities to all entities
- **Tile-Based Interest Management**

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

With a "Cloud" environment

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

With a "Cloud" environment

- Increased scalability

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

With a "Cloud" environment

- Increased scalability
- Pay for resources used (CPU, bandwidth, disk)

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

With a "Cloud" environment

- Increased scalability
- Pay for resources used (CPU, bandwidth, disk)
- High bandwidth costs
  - Large number of players, flocking

[1] http://us.battle.net/wow/en/status

Motivation

- Problem: highly-variable bandwidth needs
- Popular games: lots of game servers (WoW: 250 servers[1])

Without a "Cloud" environment

- Need to provision for peak
- Infrastructure will generally be "under-used"
- Underprovisioning: game can become unplayable in case of sudden load
- DynFilter: limiting bandwidth use to planned capacity

With a "Cloud" environment

- Increased scalability
- Pay for resources used (CPU, bandwidth, disk)
- High bandwidth costs
  - Large number of players, flocking
- DynFilter: limiting bandwidth use in order to limit costs

[1] http://us.battle.net/wow/en/status

DynFilter Architecture

1 Introduction and Background

2 DynFilter Architecture

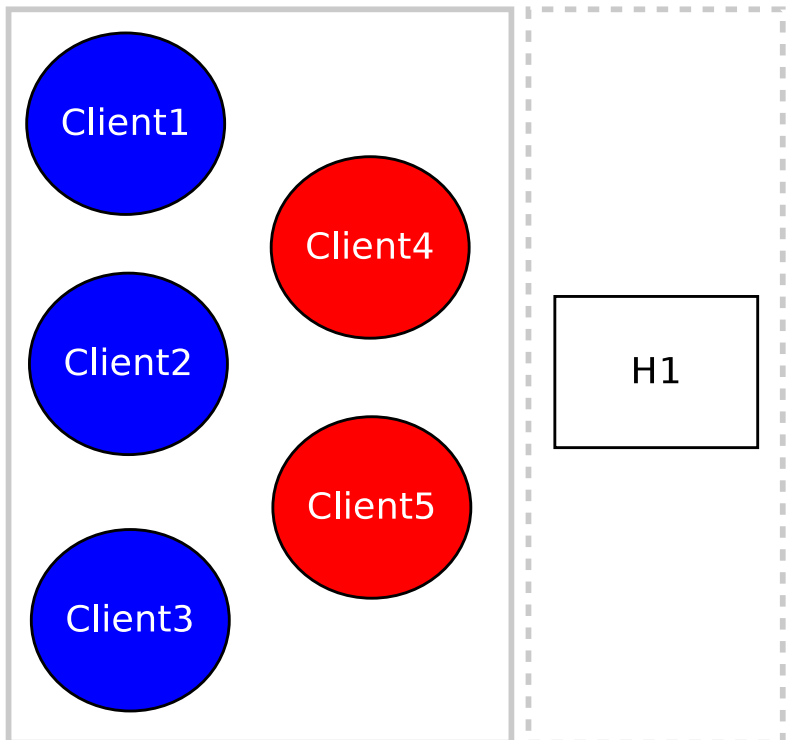
3 Load Analysis & Optimization

4 Experiments

5 Conclusion

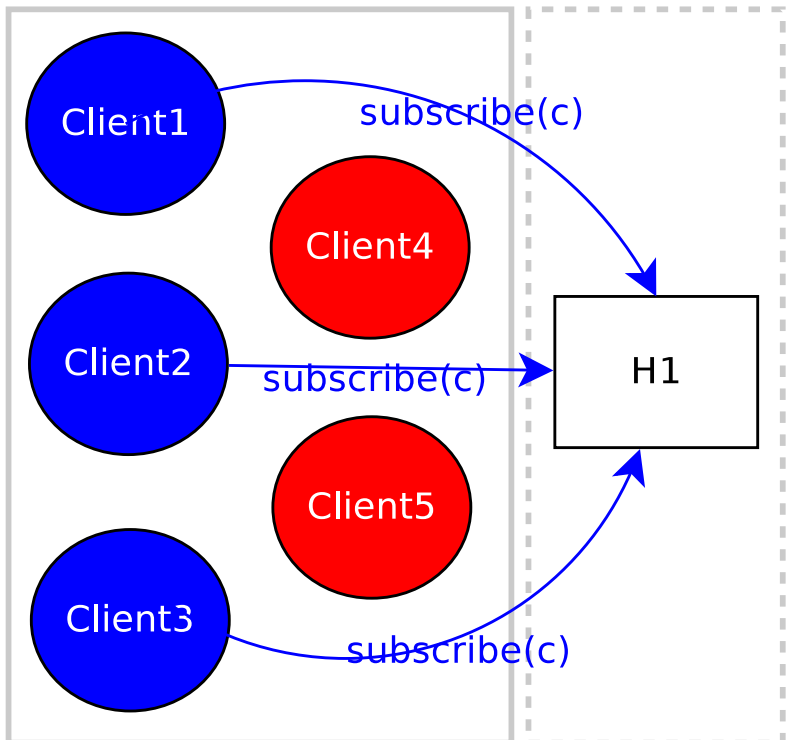
Topic-Based Publish/Subscribe

- **Subscribers** (in blue) subscribe to topics
- **Publishers** (in red) publish to topics
- All subscribers of a given topic c will receive all publications sent through c



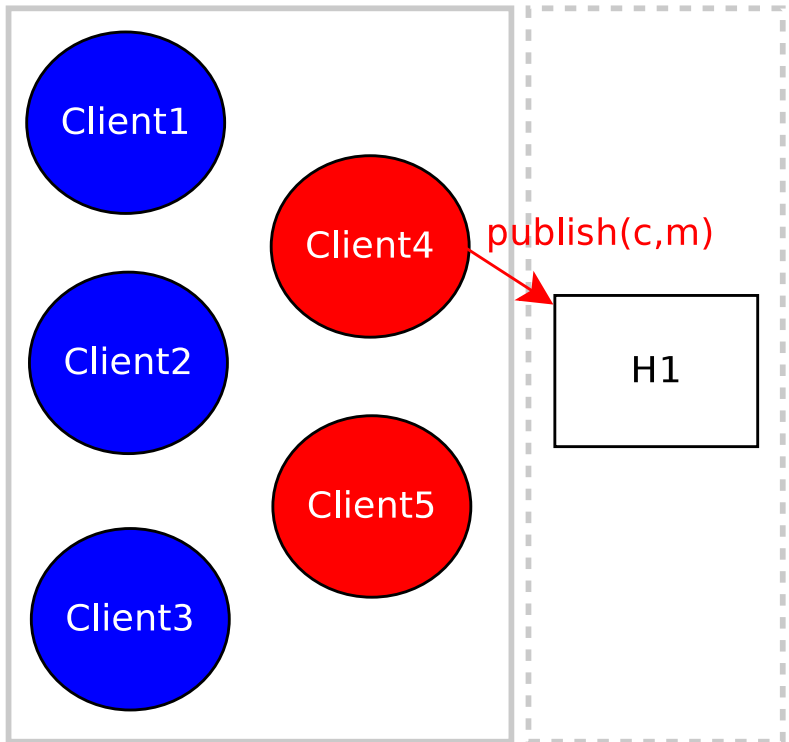
Topic-Based Publish/Subscribe

- **Subscribers** (in blue) subscribe to topics
- **Publishers** (in red) publish to topics
- All subscribers of a given topic c will receive all publications sent through c



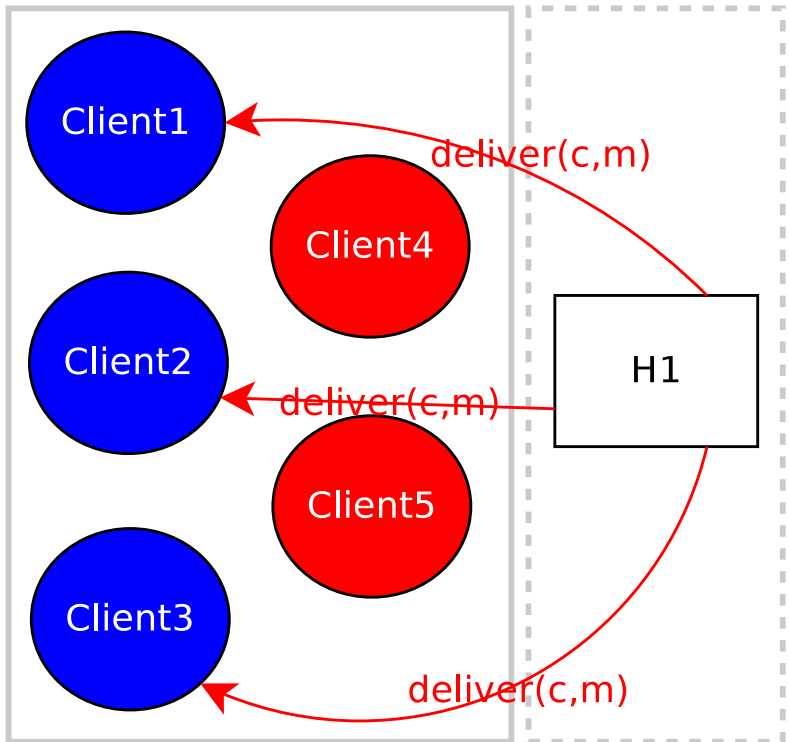
Topic-Based Publish/Subscribe

- **Subscribers** (in blue) subscribe to topics
- **Publishers** (in red) publish to topics
- All subscribers of a given topic c will receive all publications sent through c



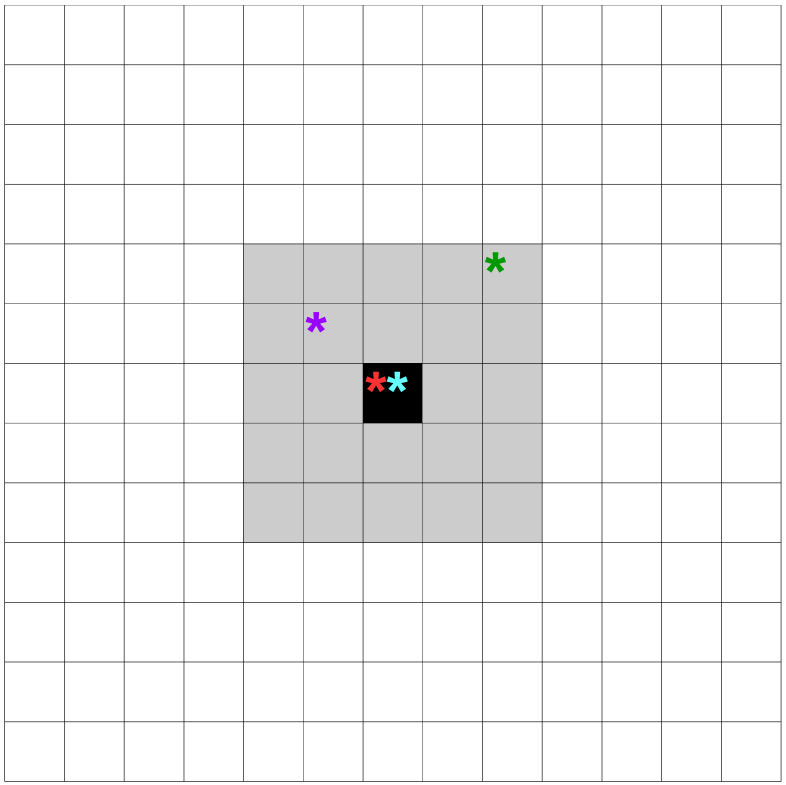
Topic-Based Publish/Subscribe

- **Subscribers** (in blue) subscribe to topics
- **Publishers** (in red) publish to topics
- All subscribers of a given topic c will receive all publications sent through c



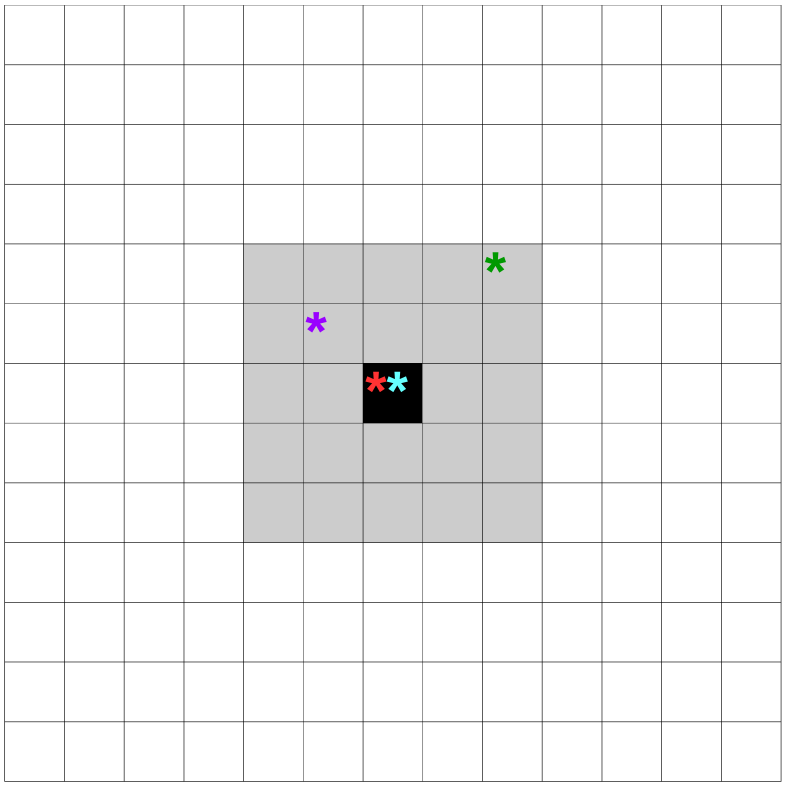
Tile-based Area-of-Interest / Subscriptions (1)

Game world divided in square tiles ($Z = 2$)



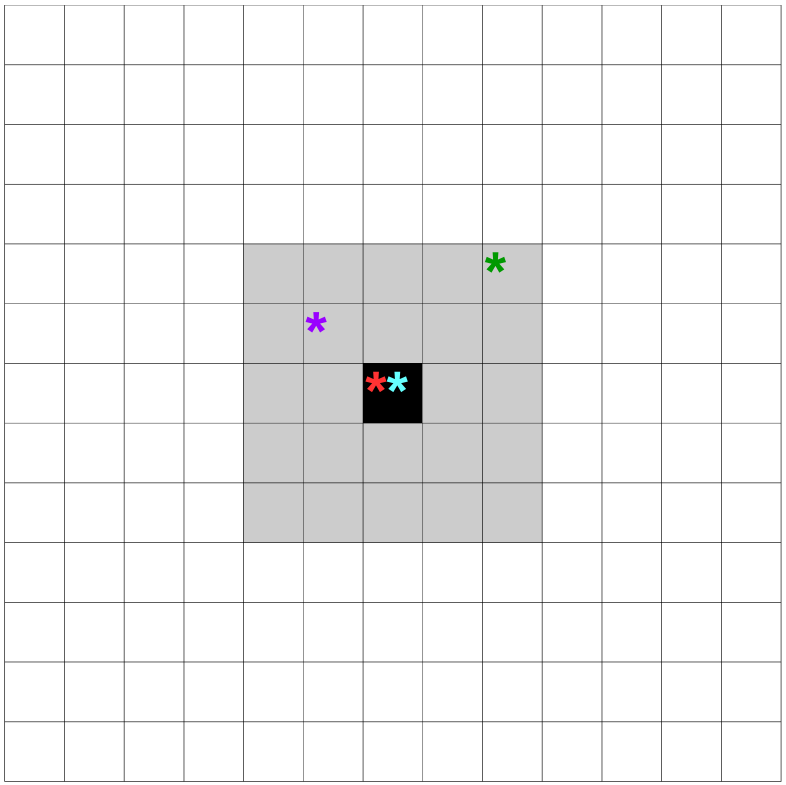
- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)

Tile-based Area-of-Interest / Subscriptions (1)



- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)
- Assuming player P is in T_{x_p,y_p} (black tile)

Tile-based Area-of-Interest / Subscriptions (1)

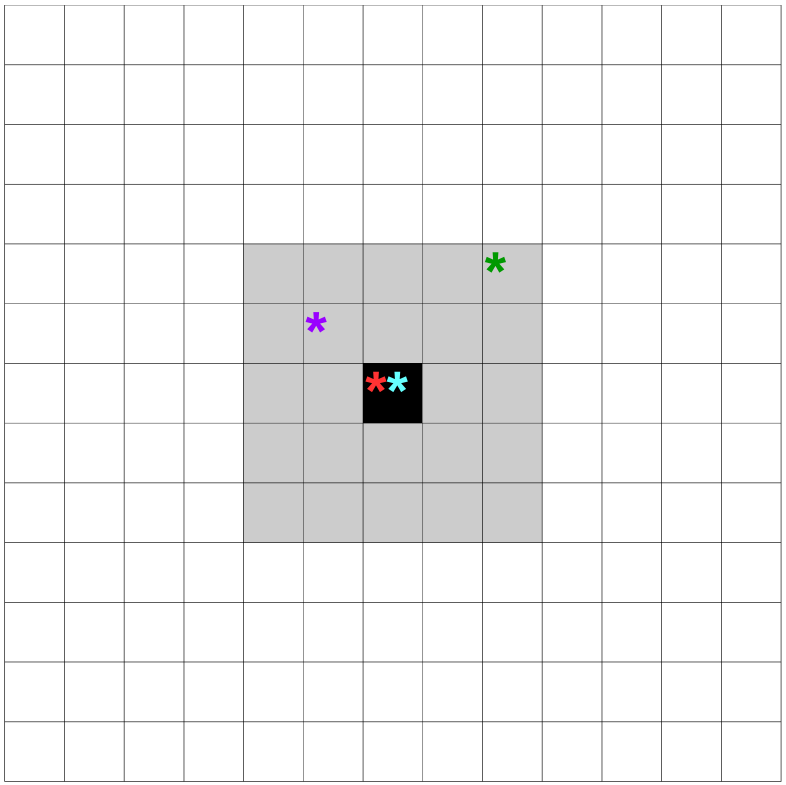


Game world divided in square tiles ($Z = 2$)

- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)
- Assuming player P is in T_{x_p,y_p} (black tile)
- Z: subscription range

Tile-based Area-of-Interest / Subscriptions (1)

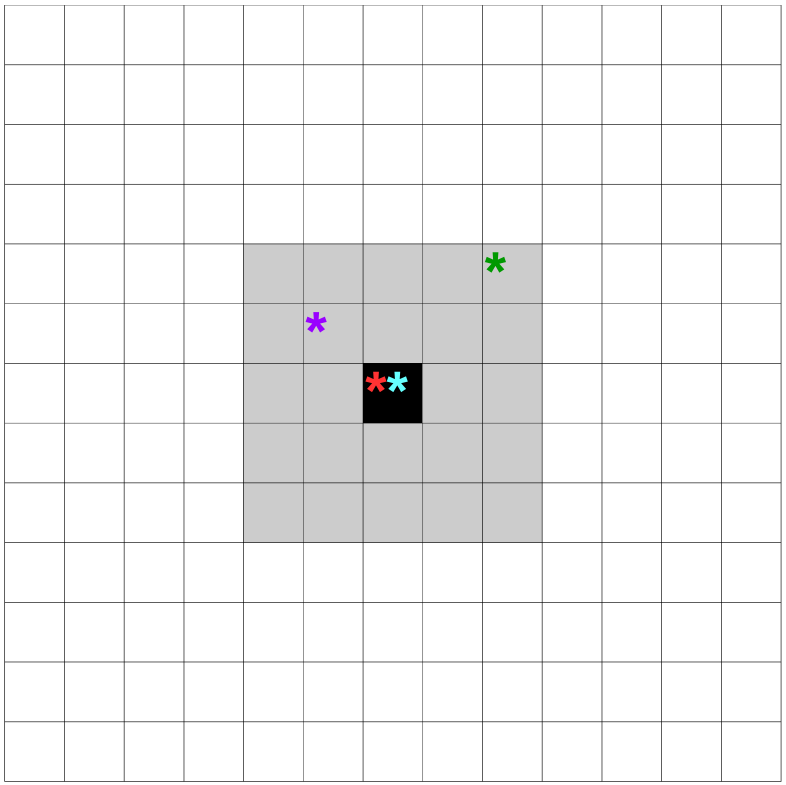
Game world divided in square tiles ($Z = 2$)



- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)
- Assuming player P is in T_{x_p,y_p} (black tile)
- Z: subscription range
- P will publish to T_{x_p,y_p}

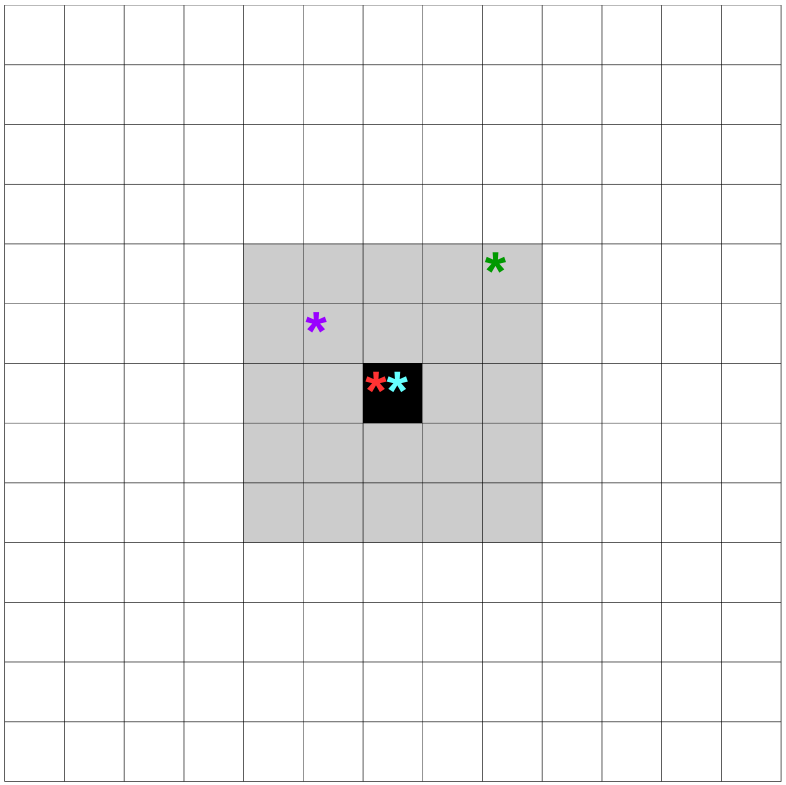
Tile-based Area-of-Interest / Subscriptions (1)

Game world divided in square tiles ($Z = 2$)



- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)
- Assuming player P is in T_{x_p,y_p} (black tile)
- Z: subscription range
- P will publish to T_{x_p,y_p}
- P will subscribe to an area of tiles $T_{x,y} | x \in \{x_p - Z, \ldots, x_p + Z\}, y \in \{y_p - Z, \ldots, y_p + Z\}$ (grey tiles)

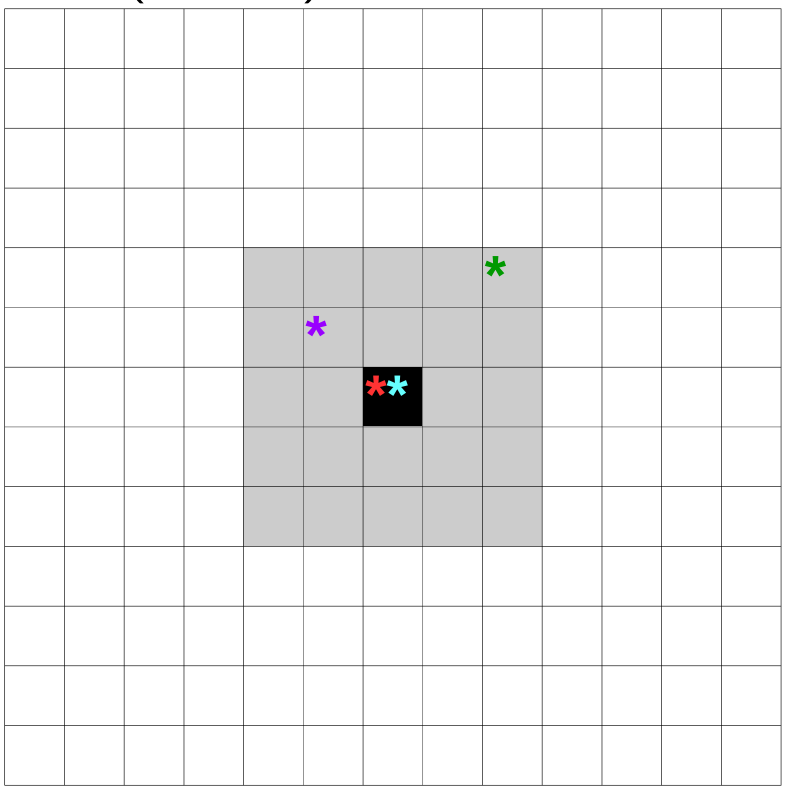
Tile-based Area-of-Interest / Subscriptions (1)



Game world divided in square tiles ($Z = 2$)

- Assuming X columns and Y rows: we have XY tiles ($T_{x,y}$)
- Assuming player P is in T_{x_p,y_p} (black tile)
- Z: subscription range
- P will publish to T_{x_p,y_p}
- P will subscribe to an area of tiles $T_{x,y} | x \in \{x_p - Z, \ldots, x_p + Z\}, y \in \{y_p - Z, \ldots, y_p + Z\}$ (grey tiles)
- Players in T_{x_p,y_p}: high update frequency $\rightarrow$ no filtering!
- Players in grey area: low update frequency $\rightarrow$ filtering may apply!

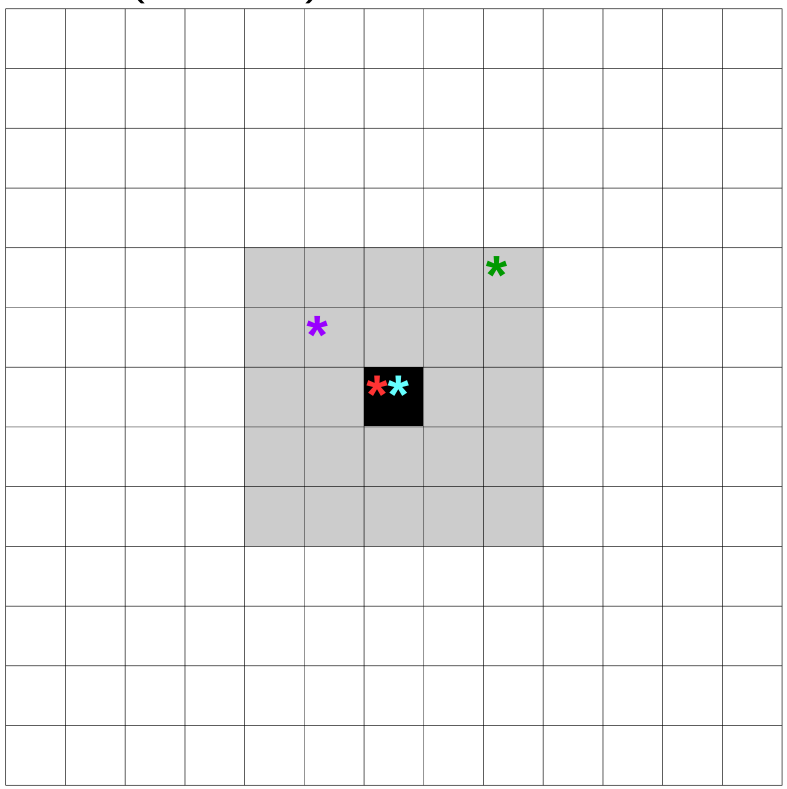
Tile-based Area-of-Interest / Subscriptions (2)



Game world divided in square tiles ($Z = 2$)

Pub/Sub → Tile-based Model

Tile-based Area-of-Interest / Subscriptions (2)

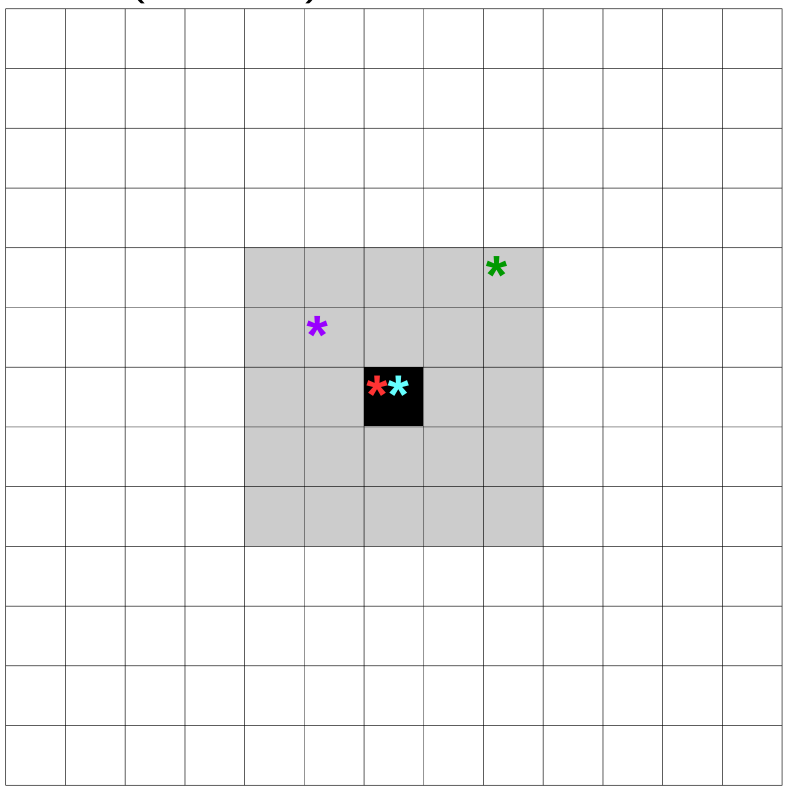


Game world divided in square tiles ($Z = 2$)

Pub/Sub → Tile-based Model

- For each tile $T_{x,y}$, we have two topics:
  - $T^H_{x,y}$: high-frequency (no filtering)
  - $T^L_{x,y}$: low-frequency (filtering can occur)

Tile-based Area-of-Interest / Subscriptions (2)

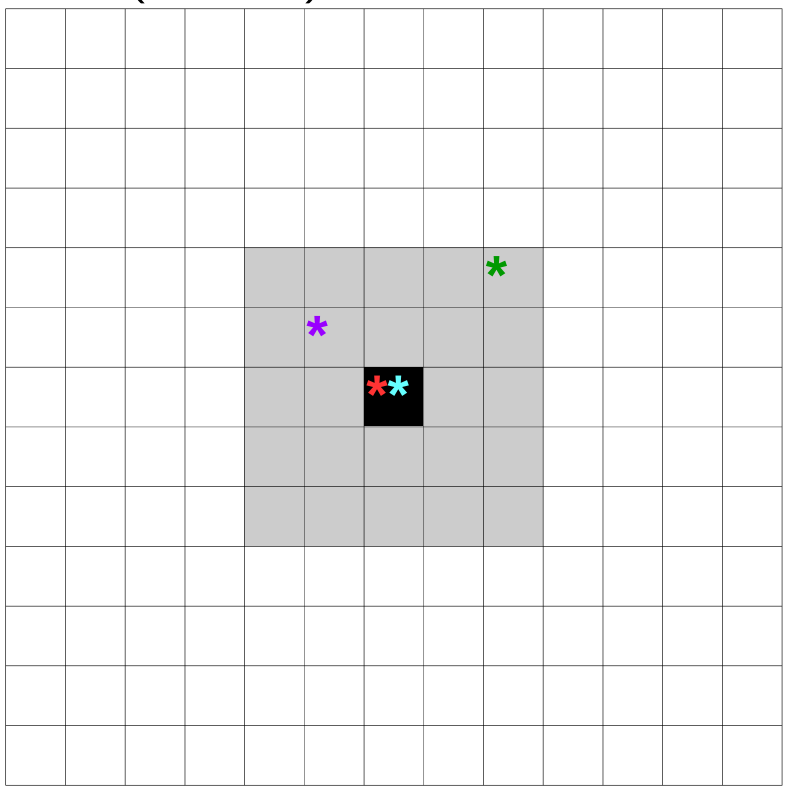


Game world divided in square tiles ($Z = 2$)

Pub/Sub → Tile-based Model

- For each tile $T_{x,y}$, we have two topics:
  - $T^H_{x,y}$: high-frequency (no filtering)
  - $T^L_{x,y}$: low-frequency (filtering can occur)
- All publications are done on $T^H_{x,y}$

Tile-based Area-of-Interest / Subscriptions (2)

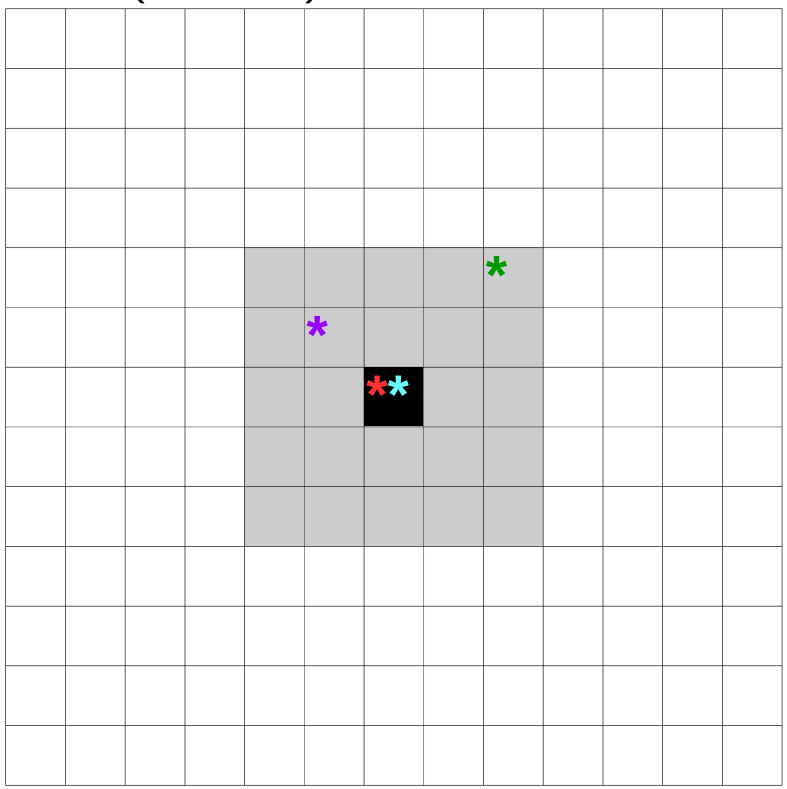


Game world divided in square tiles ($Z = 2$)

Pub/Sub → Tile-based Model

- For each tile $T_{x,y}$, we have two topics:
  - $T^H_{x,y}$: high-frequency (no filtering)
  - $T^L_{x,y}$: low-frequency (filtering can occur)
- All publications are done on $T^H_{x,y}$
- Subscriptions are done on $T^H_{x,y}$ (black) and $T^L_{x,y}$ (grey)

Tile-based Area-of-Interest / Subscriptions (2)

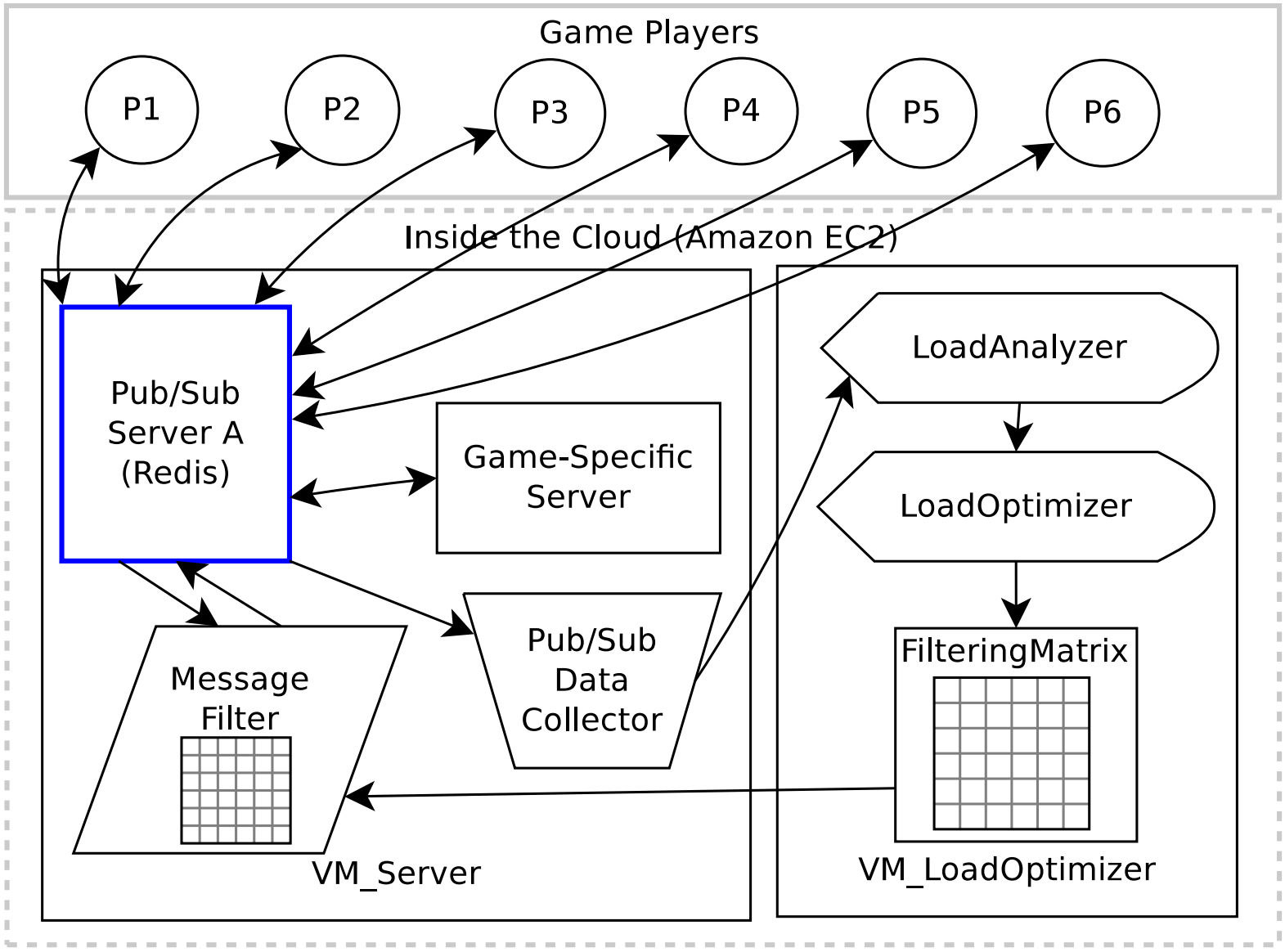


Game world divided in square tiles ($Z = 2$)

Pub/Sub → Tile-based Model

- For each tile $T_{x,y}$, we have two topics:
  - $T^H_{x,y}$: high-frequency (no filtering)
  - $T^L_{x,y}$: low-frequency (filtering can occur)
- All publications are done on $T^H_{x,y}$
- Subscriptions are done on $T^H_{x,y}$ (black) and $T^L_{x,y}$ (grey)
- Publications are forwarded from $T^H_{x,y}$ to $T^L_{x,y}$

Architectural Components



Load Analysis & Optimization

1 Introduction and Background

2 DynFilter Architecture

3 Load Analysis & Optimization

4 Experiments

5 Conclusion

Load Model

- Time unit: t (20 seconds in our experiments)

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period
  - $B_{\text{remaining}} = B_{\text{quota}} - B_{\text{used}}$

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period
  - $B_{\text{remaining}} = B_{\text{quota}} - B_{\text{used}}$
  - $B_{\text{target}} = B_{\text{remaining}}/(t_{\max} - t)$ (should be consumed over the next unit)

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period
  - $B_{\text{remaining}} = B_{\text{quota}} - B_{\text{used}}$
  - $B_{\text{target}} = B_{\text{remaining}}/(t_{\max} - t)$ (should be consumed over the next unit)
  - B_{prev}: bandwidth used over last unit

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period
  - $B_{\text{remaining}} = B_{\text{quota}} - B_{\text{used}}$
  - $B_{\text{target}} = B_{\text{remaining}}/(t_{\max} - t)$ (should be consumed over the next unit)
  - B_{prev}: bandwidth used over last unit
  - If $B_{\text{prev}} \leq B_{\text{target}}$:
    - Filtering can be reduced or canceled

Load Model

- Time unit: t (20 seconds in our experiments)
- Time period: $t_{\max}$ units (10 min. in experiments $\rightarrow$ 30 units)
- B_{quota}: max. bandwidth that can be consumed over the period
- At every time unit t:
  - B_{used}: bandwidth that have been used since the beginning of the period
  - $B_{\text{remaining}} = B_{\text{quota}} - B_{\text{used}}$
  - $B_{\text{target}} = B_{\text{remaining}}/(t_{\max} - t)$ (should be consumed over the next unit)
  - B_{prev}: bandwidth used over last unit
  - If $B_{\text{prev}} \leq B_{\text{target}}$:
    - Filtering can be reduced or canceled
  - Else
    - Filtering should be increased: too much bandwidth used
    - $B_{\text{remove}} = B_{\text{prev}} - B_{\text{target}}$

Load Optimization

Trivial Filtering

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)
- Filtering ratio depends on number of players in tile

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)
- Filtering ratio depends on number of players in tile

Filtering Ratio

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)
- Filtering ratio depends on number of players in tile

Filtering Ratio

- If $F_{x,y} = 0 \rightarrow$ no filtering

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)
- Filtering ratio depends on number of players in tile

Filtering Ratio

- If $F_{x,y} = 0 \rightarrow$ no filtering
- Otherwise, $F_{x,y}$: ratio of messages not transfered from $T^H_{x,y}$ to $T^L_{x,y}$

Load Optimization

Trivial Filtering

- **All tiles have the same filtering ratio!**
- Assuming no filtering in previous unit
- $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$
- If filtering was already in place:
  - Need to extrapolate bandwidth usage in all low-frequency tiles as if there was no filtering
  - Idea: invert the effects of the filtering already in place

DynFilter Filtering

- Tiles have a different filtering ratio ($F_{x,y}$)
- Filtering ratio depends on number of players in tile

Filtering Ratio

- If $F_{x,y} = 0 \rightarrow$ no filtering
- Otherwise, $F_{x,y}$: ratio of messages not transfered from $T^H_{x,y}$ to $T^L_{x,y}$
- Capped at a maximum value

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$
- Extrapolated outgoing bandwidth (over previous time unit of $T^L_{x,y}$), if no filtering was in place: $B^{*L}_{x,y} = \frac{B^L_{x,y}}{1-F_{x,y}}$

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$
- Extrapolated outgoing bandwidth (over previous time unit of $T^L_{x,y}$), if no filtering was in place: $B^{*L}_{x,y} = \frac{B^L_{x,y}}{1-F_{x,y}}$

Weight of tile $T_{x,y}$

- Density factor based on # subscribers: $D_{x,y} = \log_2 S_{x,y}$

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$
- Extrapolated outgoing bandwidth (over previous time unit of $T^L_{x,y}$), if no filtering was in place: $B^{*L}_{x,y} = \frac{B^L_{x,y}}{1-F_{x,y}}$

Weight of tile $T_{x,y}$

- Density factor based on # subscribers: $D_{x,y} = \log_2 S_{x,y}$
- Weight of tile $T_{x,y}$: $W_{x,y} = (B^H_{x,y} + B^{*L}_{x,y}) \cdot D_{x,y}$

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$
- Extrapolated outgoing bandwidth (over previous time unit of $T^L_{x,y}$), if no filtering was in place: $B^{*L}_{x,y} = \frac{B^L_{x,y}}{1-F_{x,y}}$

Weight of tile $T_{x,y}$

- Density factor based on # subscribers: $D_{x,y} = \log_2 S_{x,y}$
- Weight of tile $T_{x,y}$: $W_{x,y} = (B^H_{x,y} + B^{*L}_{x,y}) \cdot D_{x,y}$

Computing the Filtering Ratio

- # of bytes to "save": $Q_{x,y} = (W_{x,y}/(\sum W_{x,y})) \cdot B_{\text{remove}}$

DynFilter: Computing the Filtering Ratio

Idea: determine how many bytes we need to "save" for each $T^L_{x,y}$

Outgoing Bandwidth

- $B^H_{x,y}$, $B^L_{x,y}$: out. bandwidth over prev. unit of $T^H_{x,y}$, $T^L_{x,y}$
- Extrapolated outgoing bandwidth (over previous time unit of $T^L_{x,y}$), if no filtering was in place: $B^{*L}_{x,y} = \frac{B^L_{x,y}}{1-F_{x,y}}$

Weight of tile $T_{x,y}$

- Density factor based on # subscribers: $D_{x,y} = \log_2 S_{x,y}$
- Weight of tile $T_{x,y}$: $W_{x,y} = (B^H_{x,y} + B^{*L}_{x,y}) \cdot D_{x,y}$

Computing the Filtering Ratio

- # of bytes to "save": $Q_{x,y} = (W_{x,y}/(\sum W_{x,y})) \cdot B_{\text{remove}}$
- Filtering ratio: $F_{x,y} = \frac{Q_{x,y}}{B^{*L}_{x,y}}$

Experiments

1 Introduction and Background

2 DynFilter Architecture

3 Load Analysis & Optimization

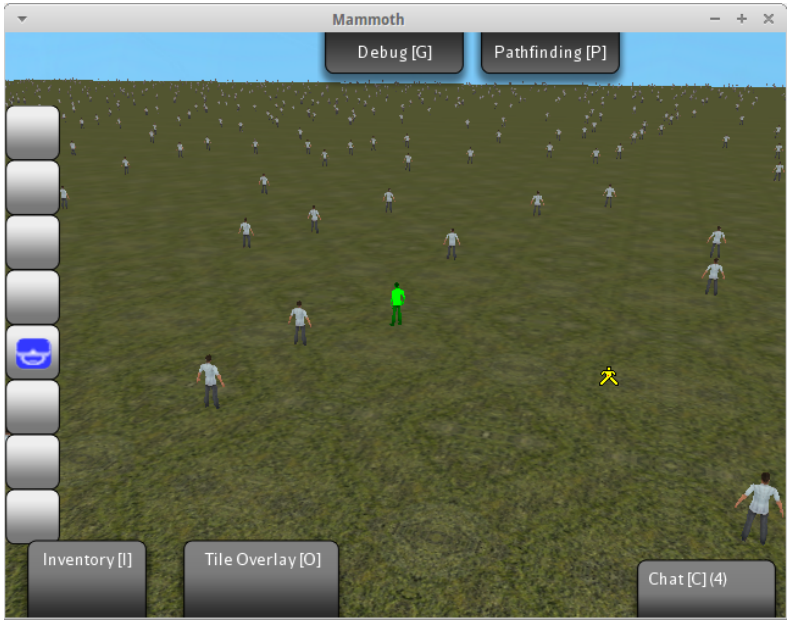
4 Experiments

5 Conclusion

Implementation & Experimental Setup

Implementation

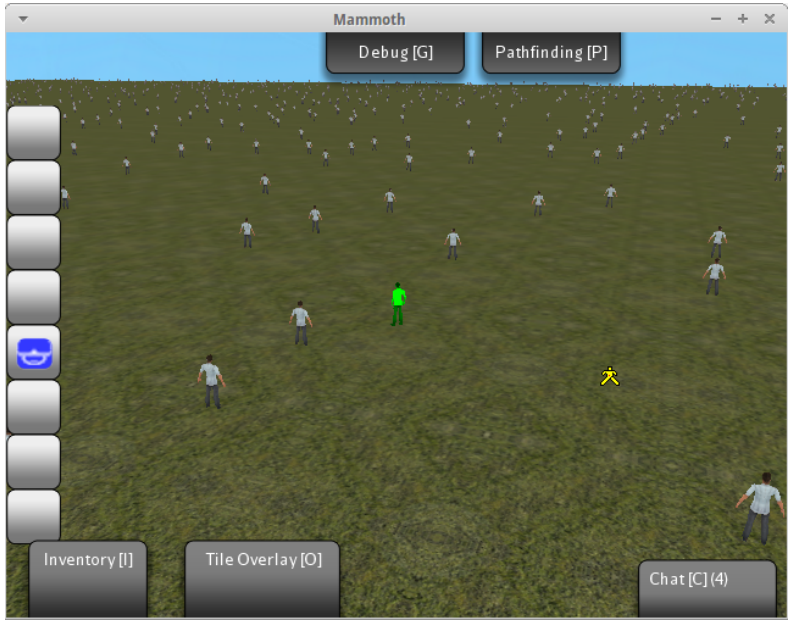
- Implemented in Java, on top of Dynamoth
- Pub/Sub: unmodified open-source Redis middleware
- Experiments run on DynGame (prototype game skeletton built on top of Mammoth)
- DynGame: large amount of AI-controlled players (random-waypoint)
- Supports flocking



Implementation & Experimental Setup

Experimental Setup

- 20 Amazon EC2 instances
- 15 players per instance (max 250)
- $Z = 2$ (subscription to up to 25 tiles)
- Period of 10 minutes
- Units of 20 seconds



Experiments

Experiment 1: FPS Game / Scalability

- Scalability in a FPS-like game with many players
- Very high frequency of updates (20 updates/sec)
- Up to 150 players (Q3=16, WatchMen=48)
- 10x10 map (100 tiles)
- Player can view up to 25% of the map ($Z = 2$)
- Bandwidth alloc.: 8000Mb

Experiments

Experiment 1: FPS Game / Scalability

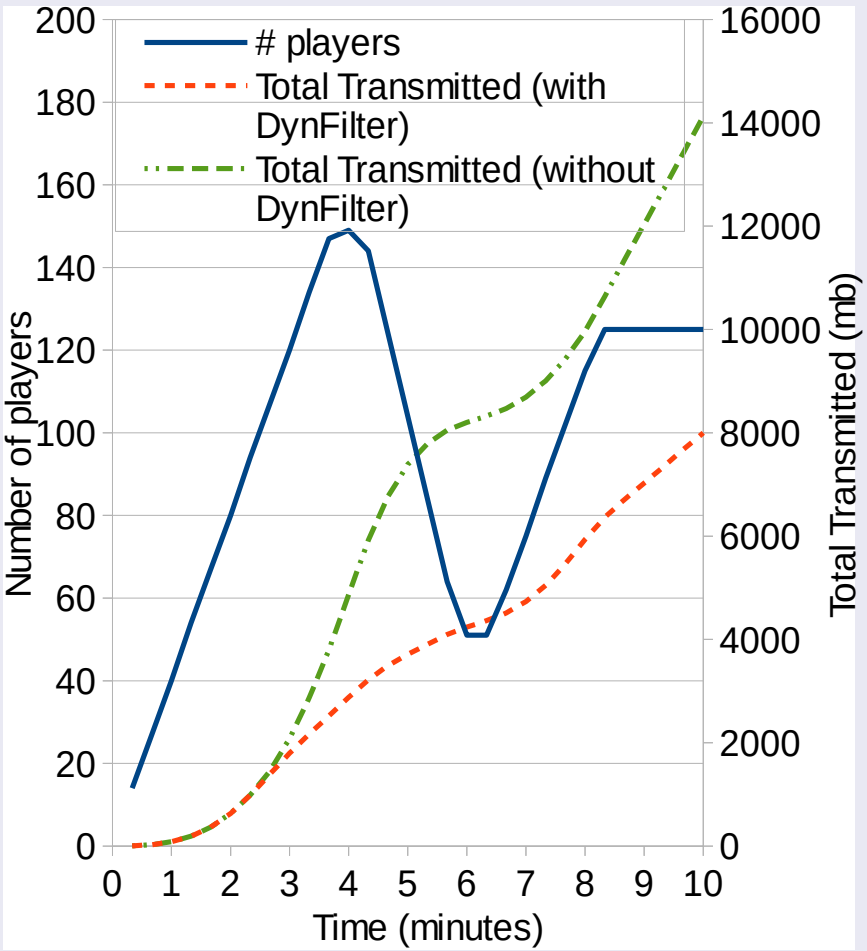
- Scalability in a FPS-like game with many players
- Very high frequency of updates (20 updates/sec)
- Up to 150 players (Q3=16, WatchMen=48)
- 10x10 map (100 tiles)
- Player can view up to 25% of the map ($Z = 2$)
- Bandwidth alloc.: 8000Mb

Experiment 2: MMORPG Game with Flocking

- Flocking in medium-scale MMOGs
- Flocking: quadratic growth in message delivery
- Up to 250 players
- 20x20 map (400 tiles)
- Player can view 6.2% of the map ($Z = 2$)
- Flocking ratio ψ between 0 and 0.5
- Flocking: 4x4 centric tiles
- Bandwidth alloc.:10000Mb

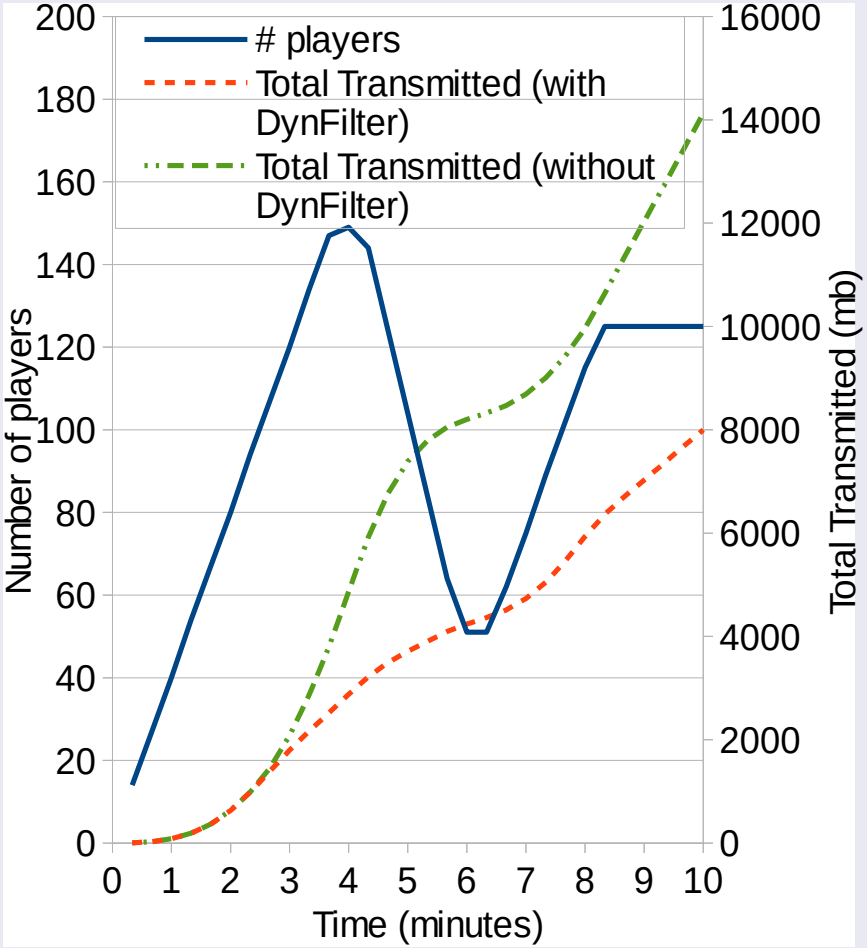
FPS Game - Results (1)

Number of Players and Total Outgoing Bandwidth

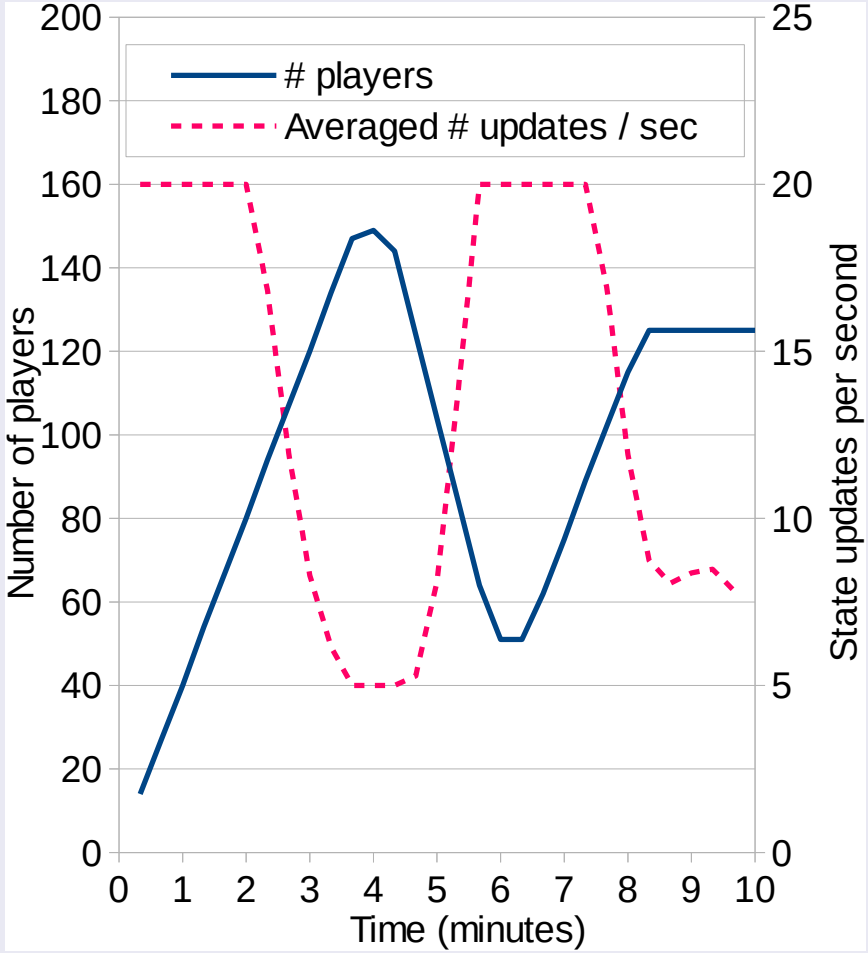


FPS Game - Results (1)

Number of Players and Total Outgoing Bandwidth



State Updates per Second

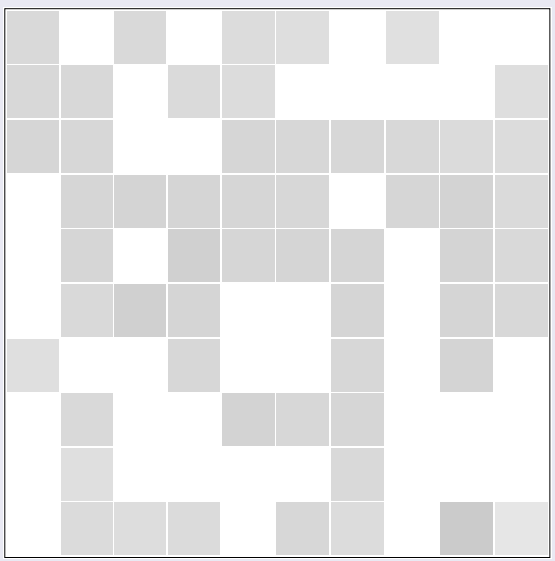


FPS Game - Results (2)

→ Bandwidth savings of 43%.

Filtering Ratio Heat Maps:

$t = 2.3$; $\hat{F} = 0.14$

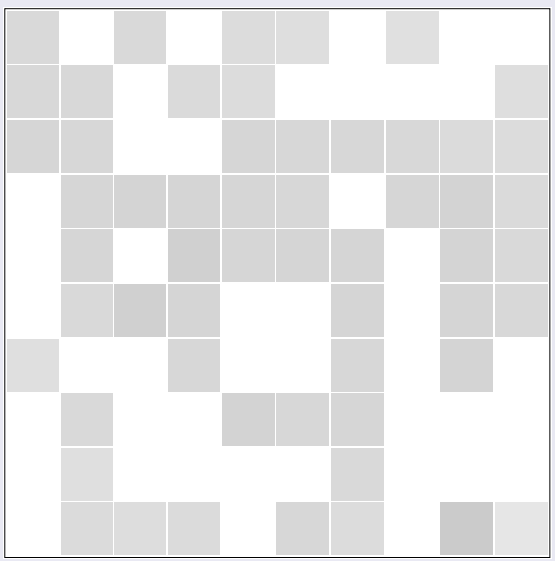


FPS Game - Results (2)

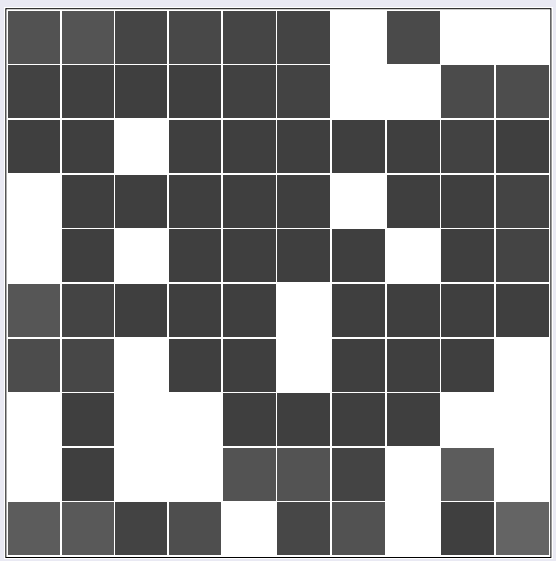
→ Bandwidth savings of 43%.

Filtering Ratio Heat Maps:

$t = 2.3$; $\hat{F} = 0.14$



$t = 4$; $\hat{F} = 0.73$

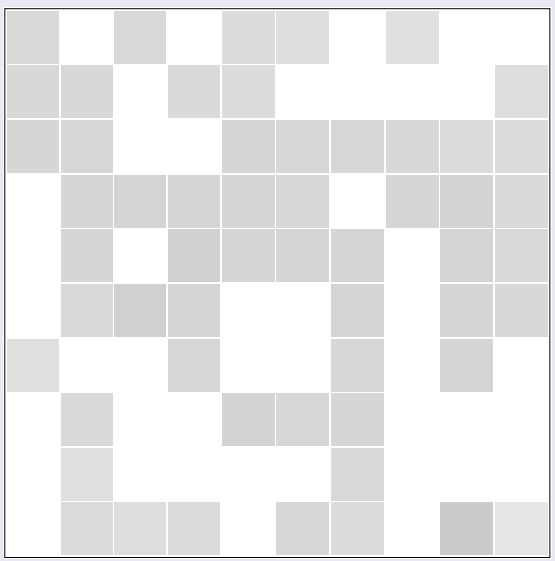


FPS Game - Results (2)

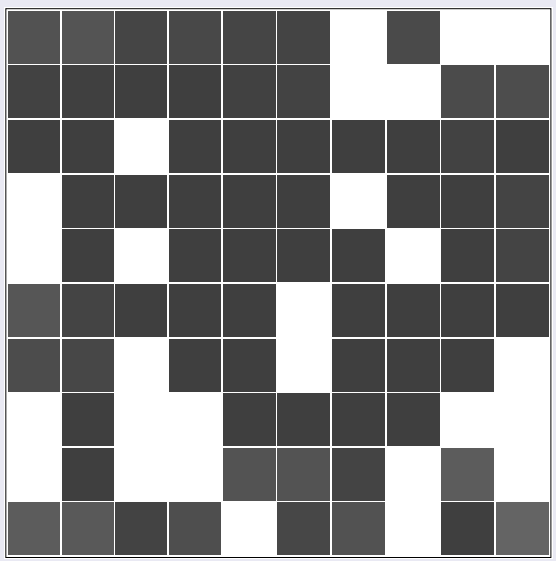
→ Bandwidth savings of 43%.

Filtering Ratio Heat Maps:

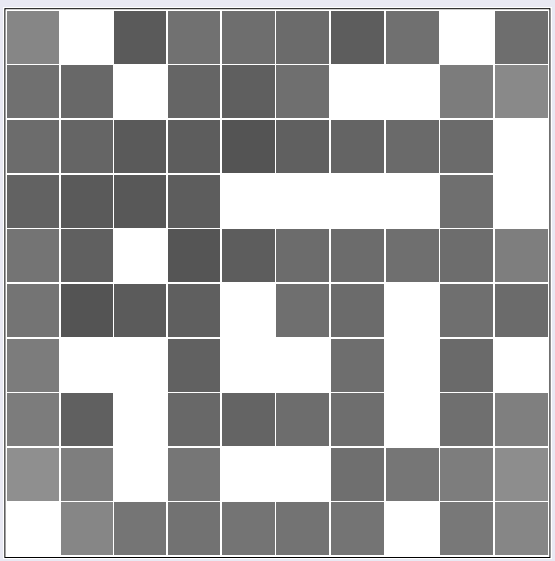
$t = 2.3$; $\hat{F} = 0.14$



$t = 4$; $\hat{F} = 0.73$

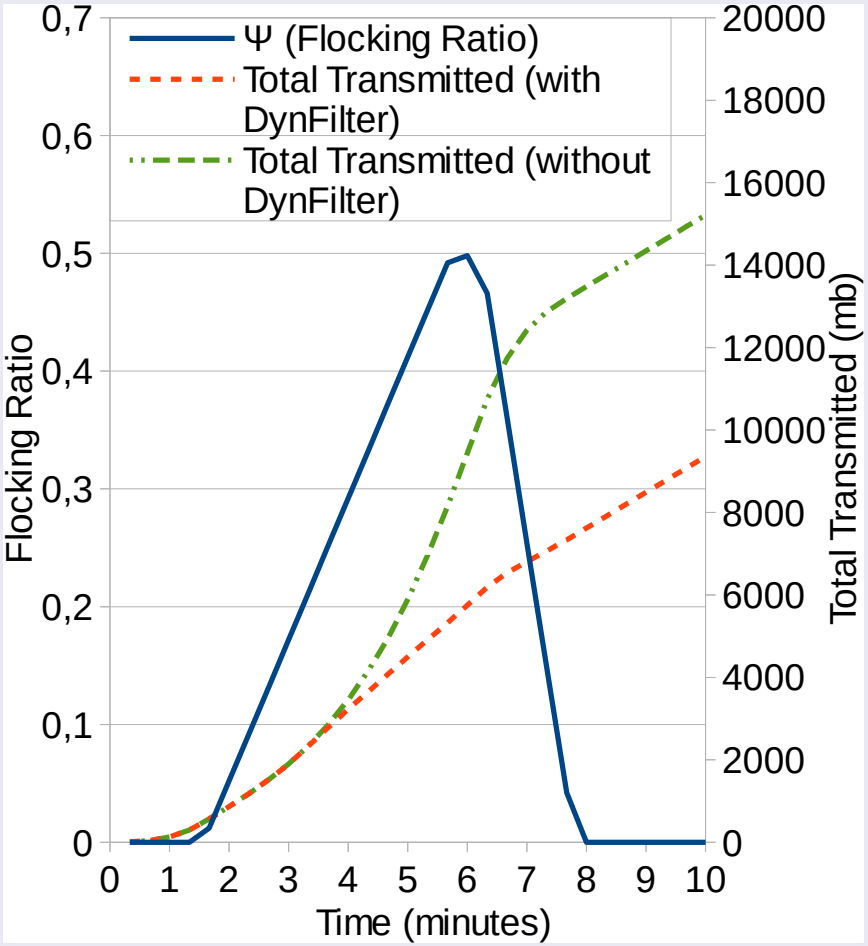


$t = 8.7$; $\hat{F} = 0.54$



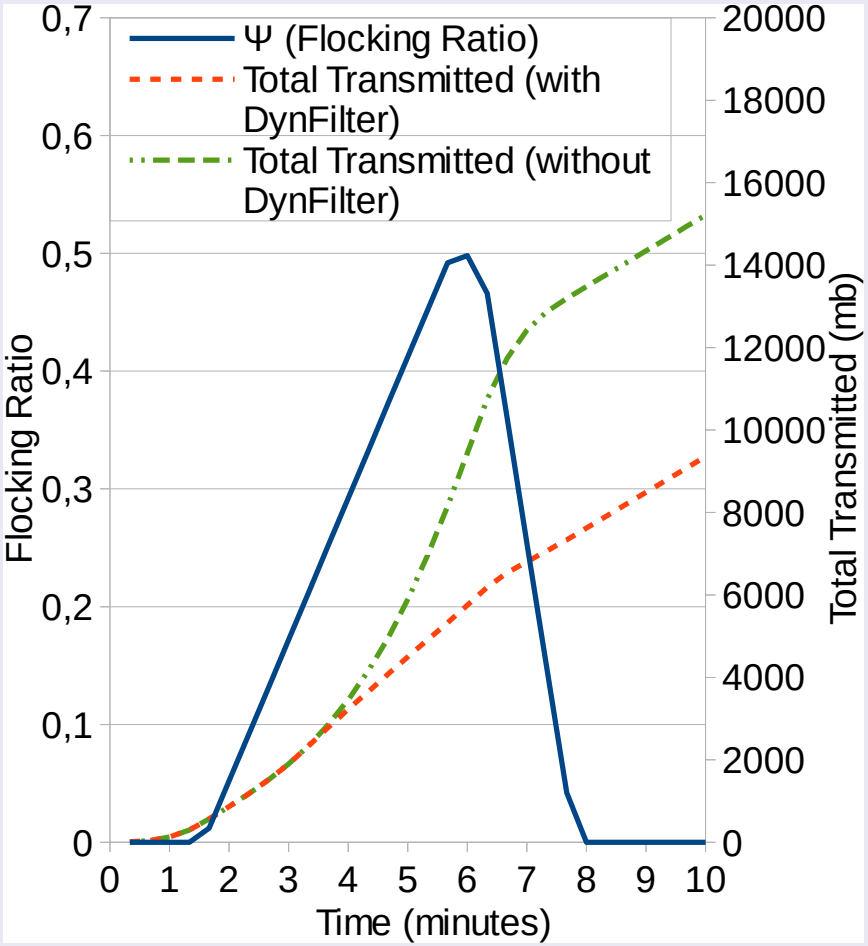
MMORPG - Results (1)

Flocking Ratio and Total Outgoing Bandwidth

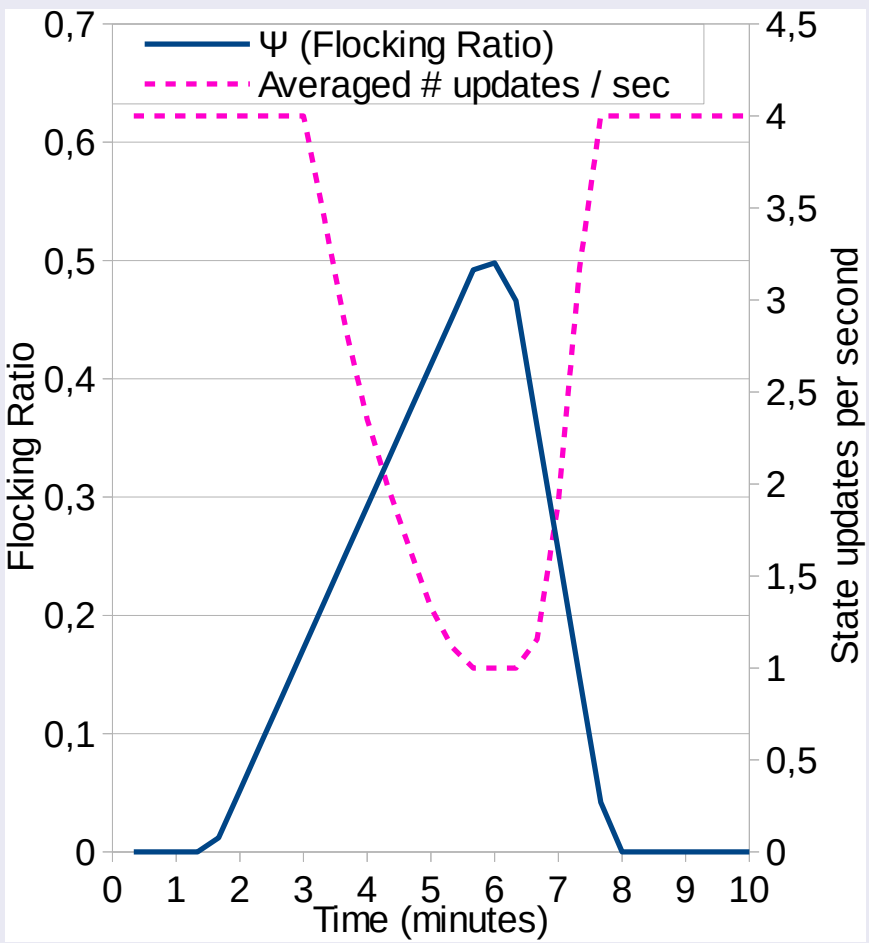


MMORPG - Results (1)

Flocking Ratio and Total Outgoing Bandwidth



State Updates per Second

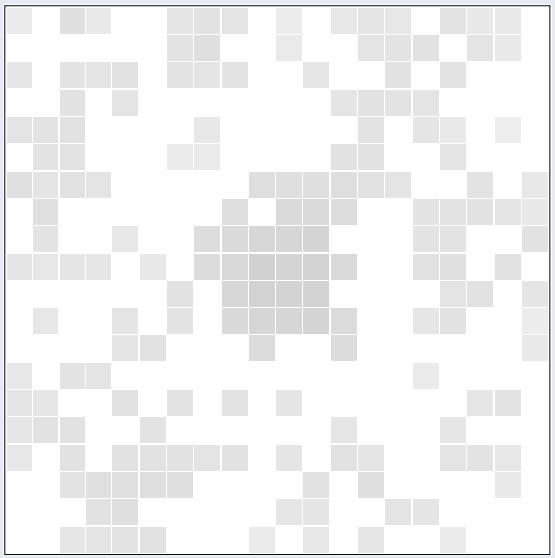


MMORPG - Results (2)

→ Bandwidth savings of 38%.

Filtering Ratio Heat Maps:

$t = 3.5$; $\psi = 0.21$

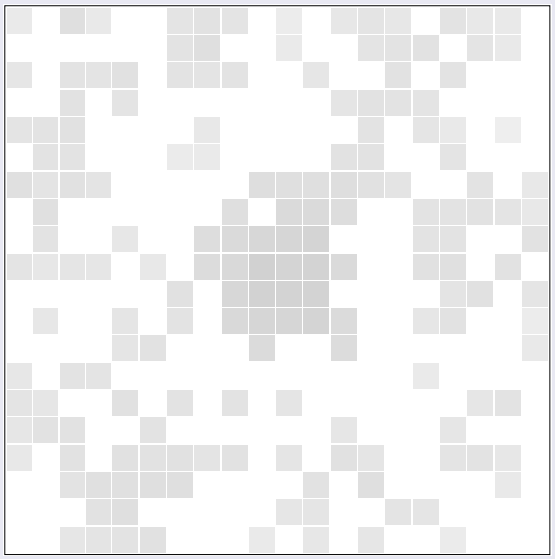


MMORPG - Results (2)

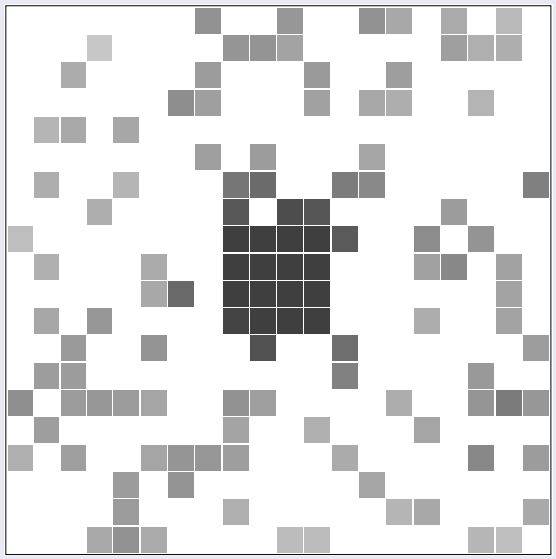
→ Bandwidth savings of 38%.

Filtering Ratio Heat Maps:

$t = 3.5$; $\psi = 0.21$



$t = 6$; $\psi = 0.50$

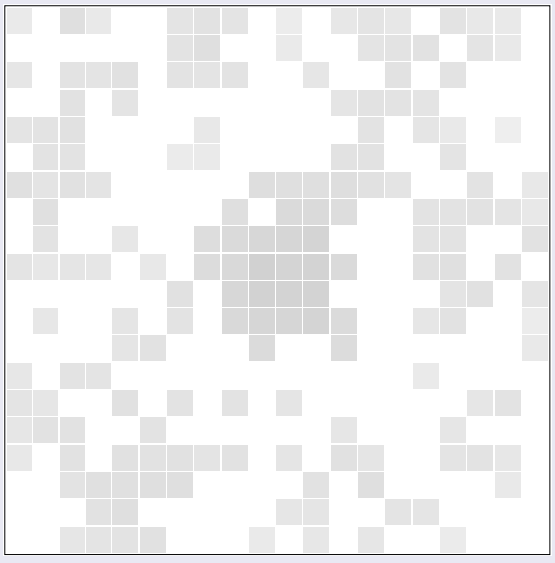


MMORPG - Results (2)

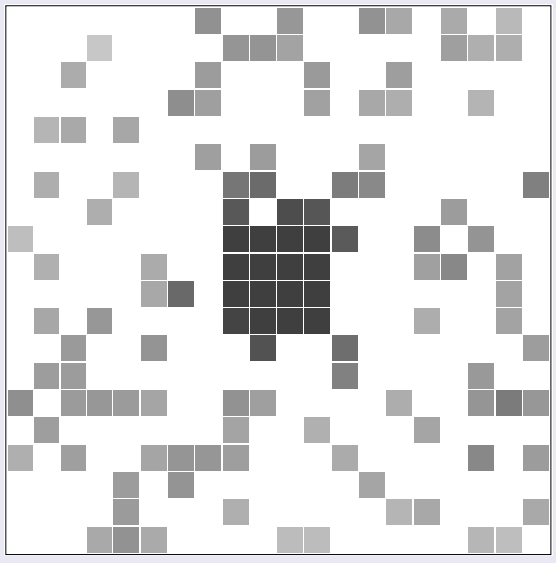
→ Bandwidth savings of 38%.

Filtering Ratio Heat Maps:

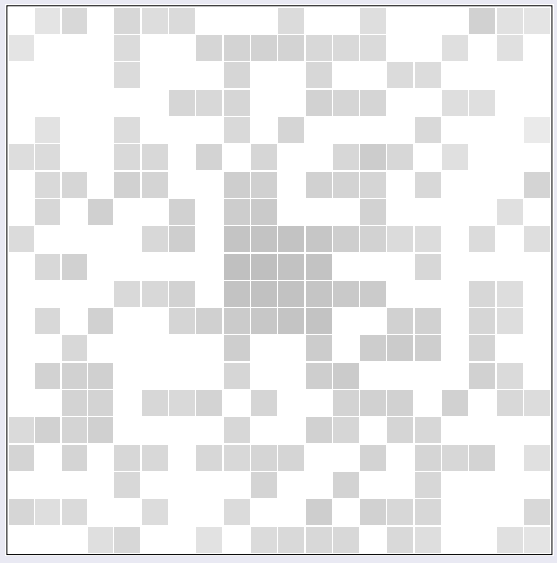
$t = 3.5$; $\psi = 0.21$



$t = 6$; $\psi = 0.50$



$t = 7.3$; $\psi = 0.15$



Conclusion & Future Work

1 Introduction and Background

2 DynFilter Architecture

3 Load Analysis & Optimization

4 Experiments

5 Conclusion

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting
- Meeting predefined quotas

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting
- Meeting predefined quotas
- Full state updating for close entities

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting
- Meeting predefined quotas
- Full state updating for close entities
- Per-tile filtering (adapts to volume of players in tiles)

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting
- Meeting predefined quotas
- Full state updating for close entities
- Per-tile filtering (adapts to volume of players in tiles)
- Experiments: FPS and MMORPG

Conclusion

- DynFilter: middleware designed to adaptively filter game state update messages
- Limiting bandwidth use within games in a Cloud setting
- Meeting predefined quotas
- Full state updating for close entities
- Per-tile filtering (adapts to volume of players in tiles)
- Experiments: FPS and MMORPG
- Important bandwidth savings while maintaining a minimal update frequency

Future Work

- N-Layered Filtering

Future Work

- N-Layered Filtering
- Quality of Experience evaluations

Future Work

- N-Layered Filtering
- Quality of Experience evaluations
- Classification of game messages (only some messages could be dropped)
  - Can exploit to the Pub/Sub layer for that: dropping messages from some topics only

Conclusion & Future Work

Thank you for your attention!