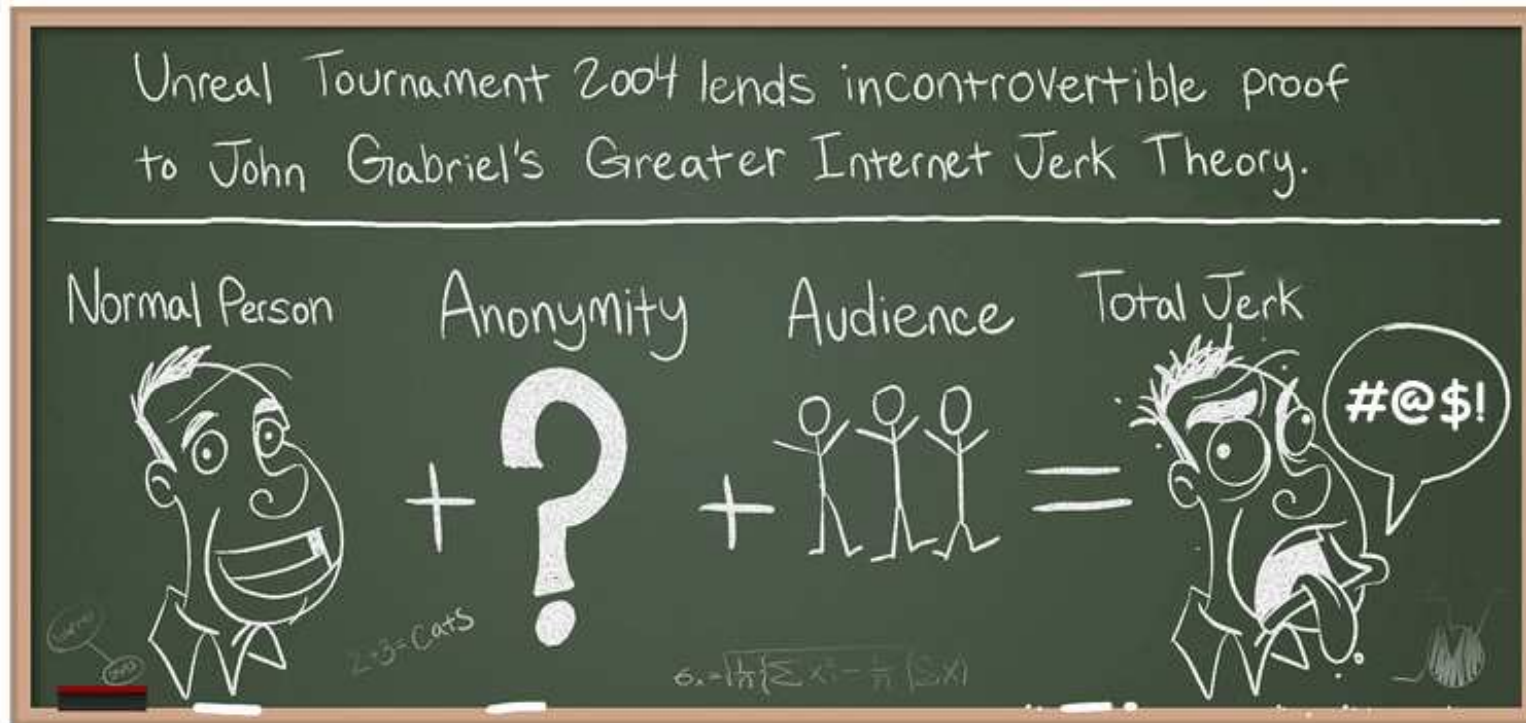


PlayerRating: A Reputation System for Multiplayer Online Games

Ed Kaiser
Wu-chang Feng



The Online Behavior Problem



Penny Arcade 03/19/2004

Motivation

- **Players misbehave**
 - premeditated (e.g., harassment, cheating, scamming, *griefing*)
 - unpremeditated (e.g., poor sportsmanship, *rage-quitting*)
- **Difficult to control antisocial behavior**
 - limited policing resources (i.e., manpower, storage)
 - hard to catch after the fact
- **Existing social tools fall short**
 - experiences are not shared (e.g., friend-list, ignore-list)
 - too discrete – only 3 options: like / dislike / unknown
 - impersonal (e.g., accumulated score systems)

Reputation Systems

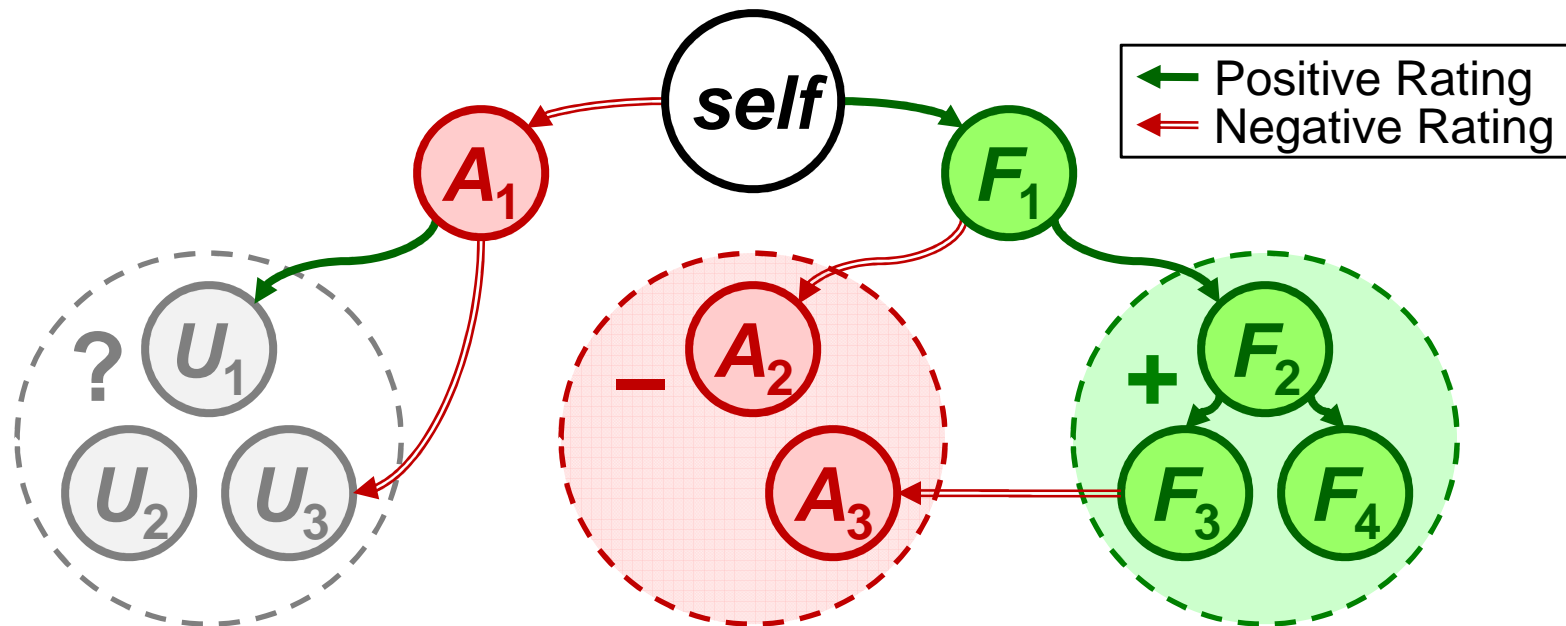
- **Share experiences**
 - avoid antisocial players encountered by peers
 - reward players for behaving well
- **Personalized view of the community**
 - based on peers whom the player likes / dislikes
- **Fine granularity**
 - affinity for peers can be sorted
 - players can be auto-grouped more accurately
 - developer can focus investigation on worst offenders

The PlayerRating Approach

- **Calculate reputation in distributed fashion**
 - minimizes server resources needed
 - yet can still be centralized
- **Incrementally deployable**
 - useful to adopters without majority adoption
 - *implemented as a World of Warcraft mod*
- **Encourages participation**
 - calculated reputations become more accurate as player creates additional accurate ratings
- **Resistant to abuse**
 - no collusion or Sybil attacks

PlayerRating Intuition

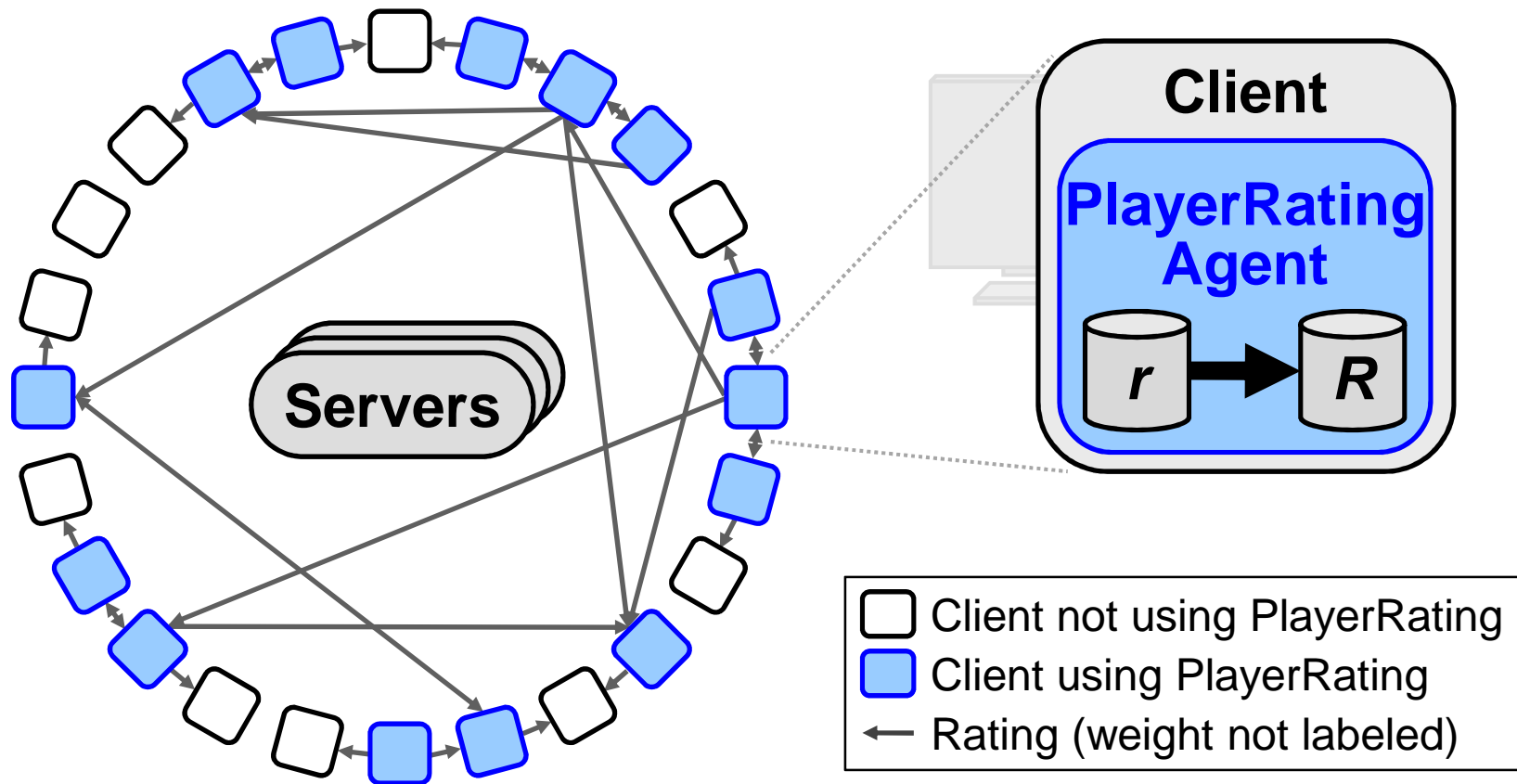
- Ratings $r_{i,j}$ and reputations $R_j \in [-1.0, 1.0]$
 - negative \rightarrow dislike, zero \rightarrow unknown, positive \rightarrow like
 - positive ratings are transitive (friend-of-friend is friend)



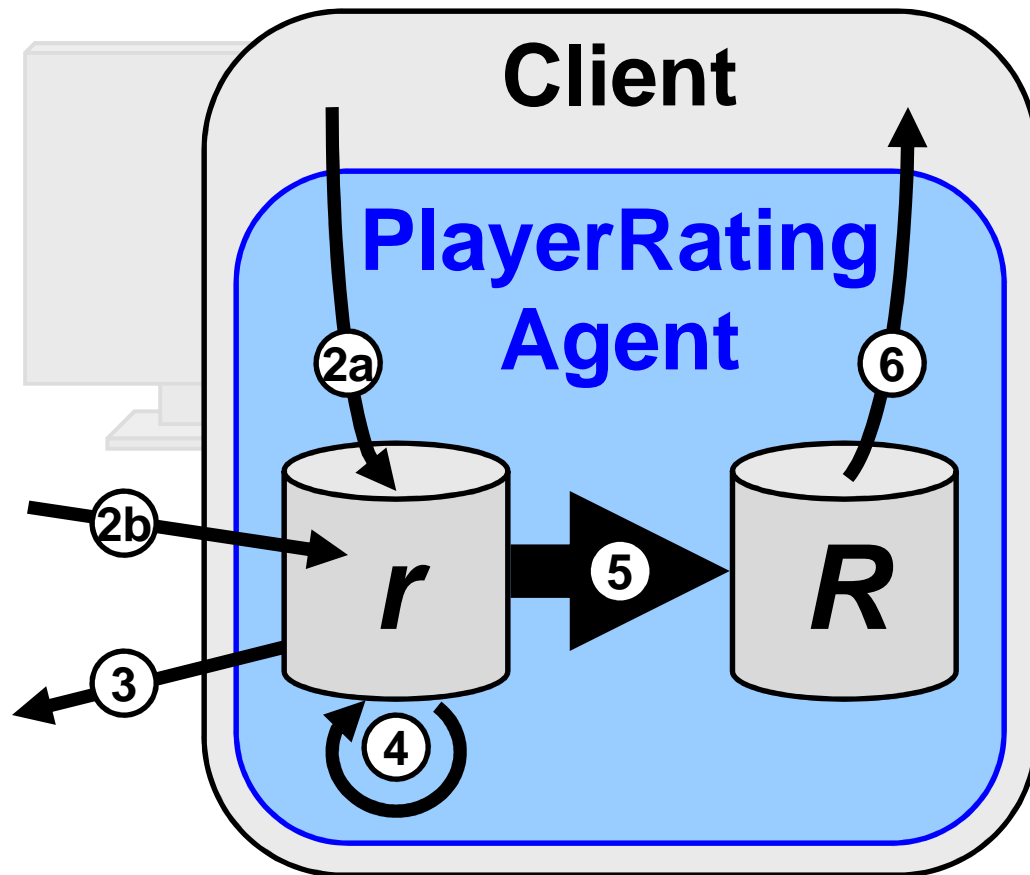
- Challenge is to reconcile conflicting ratings

PlayerRating Overview

- Community is a sparse directed graph
 - players are nodes, ratings are weighted edges



PlayerRating Design



- 1) Initialization
- 2) Set rating
 - a) locally
 - b) from peers
- 3) Distribute ratings
- 4) Expire ratings
- 5) **Calculate reputations**
- 6) Get reputation

Initialization()

- Start everything as unknown (i.e., zero)
 - ratings $r_{i,j}$
 - corresponding time-to-live $TTL_{i,j}$ values
 - reputations R_j
- Set own reputation (R_{self}) to maximum

SetRating()

- Player creates/changes ratings at any time
 - asynchronous to other system algorithms
 - unobtrusive addition to user interface



- One algorithm for inserting ratings
 - sourced locally
 - shared by one's peers
 - disallow peers from rating themselves
- Set $TTL_{i,j}$ to **TTL_MAX**
 - indicates rating is fresh

DistributeRatings ()

- Share ratings with peers
 - via in-game communication channels
 - source is authenticated by server (i.e., no spoofing)
 - prevents Sybil attacks
- According to game-specific policy
 - which may distribute ratings
 - by channel broadcast or directed
 - when rating is created/changed
 - when player finishes interacting with ratee
 - all at once
 - periodically (sequentially or randomly selected)
 - redundancy is necessary
 - peers may not be online to receive first broadcast

ExpireRatings()

- Peers' ratings should be aged and expired
 - new ratings are more relevant
 - old ratings may no longer be necessary
 - rating may have been revoked
 - rater or ratee may have quit playing
 - reduces overall state
- With period T_{expire} , decrement every TTL
 - if $TTL_{i,j}$ reaches 0, remove rating $r_{i,j}$
 - ratings will last for $T_{expire} \times TTL_MAX$
 - $T_{expire} = 1 \text{ pHour}$
 - $TTL_MAX = 24 \times 30 \rightarrow 1 \text{ pMonth}$

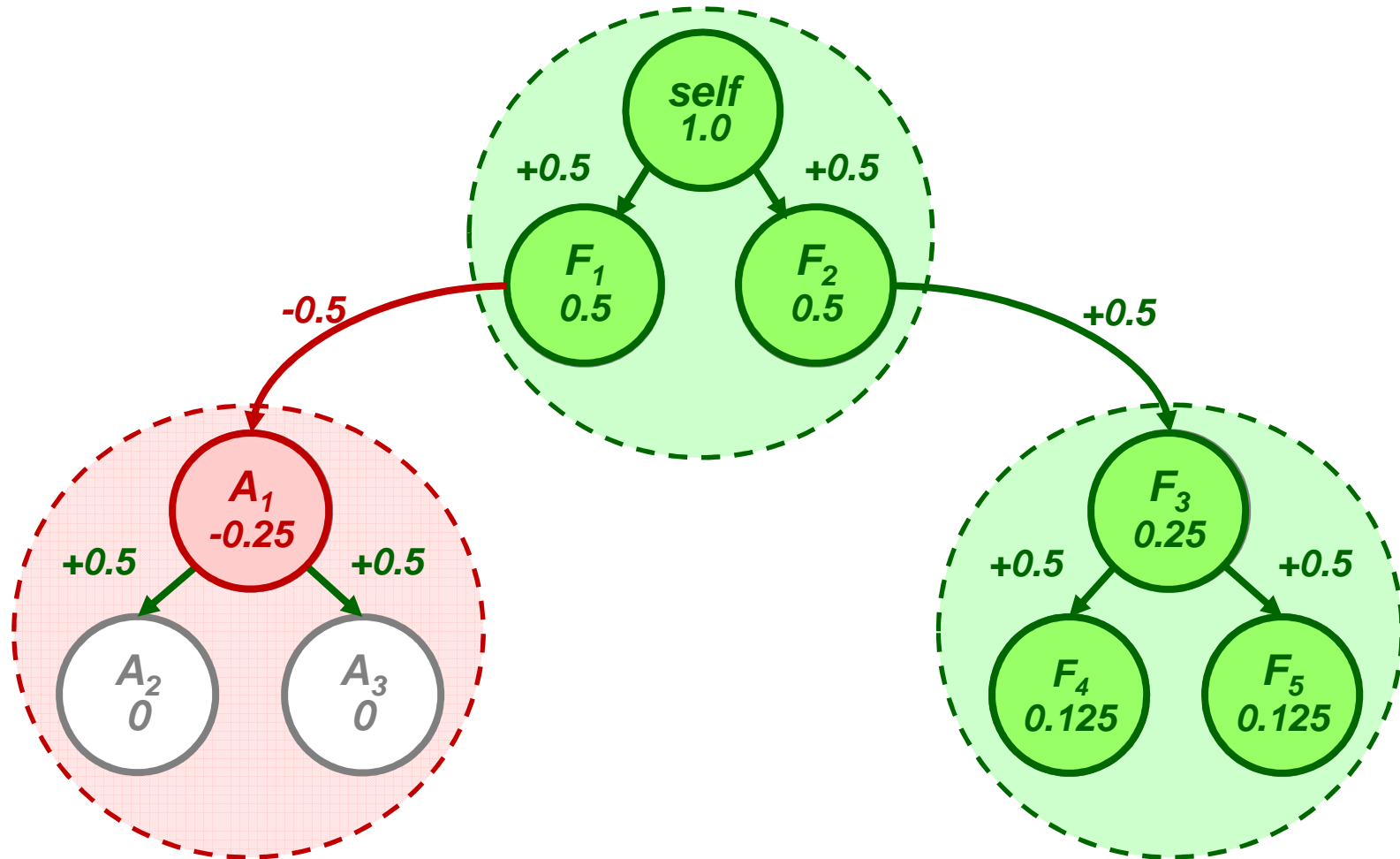
CalculateReputations()

- Reputations are the average of ratings
 - rating $r_{i,j}$ is modified by
 - rater's reputation (i.e., $\times R_i$)
 - semantically: i trusts j relative to their self
 - prevents positive feedback loops
 - rating age (i.e., $\times \text{Decay}(TTL_{i,j})$)
 - ensures newer ratings are more relevant
 - function must be non-negative and non-decreasing
 - weighted by each rater's influence
 - determined by $\text{Influence}(R_i)$

R_i	$R_i > 0$
0	otherwise

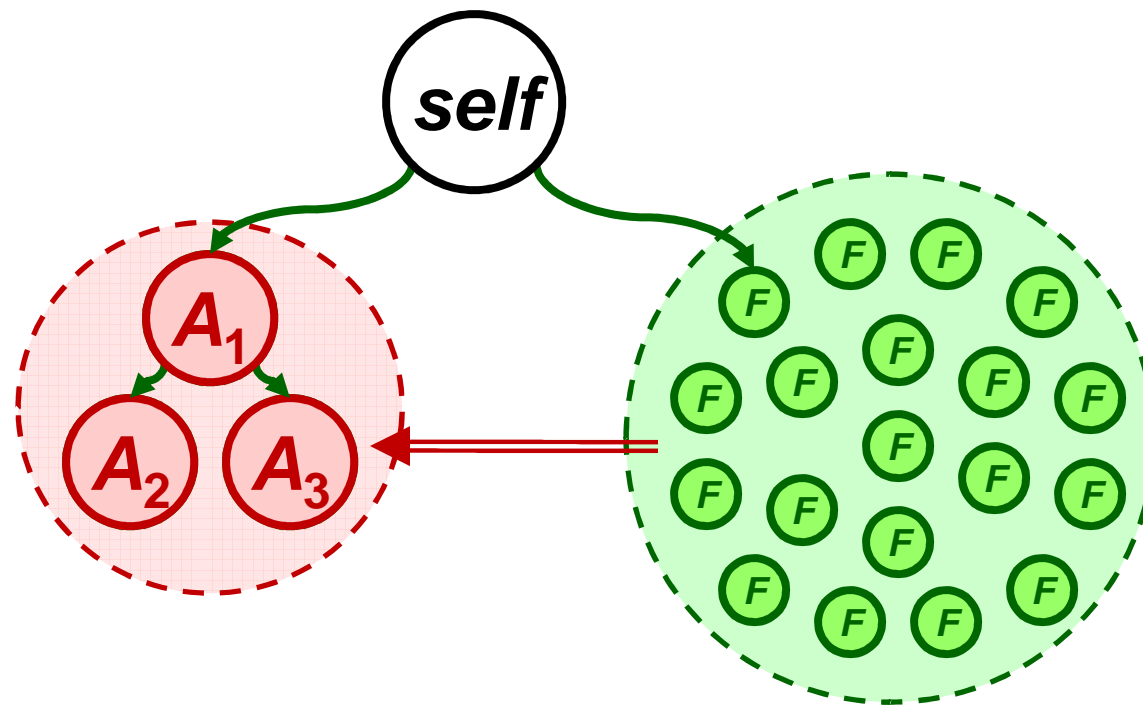
 - function must be non-negative and non-decreasing
 - must have positive influence to count
- Iterated periodically with period $T_{\text{calculate}}$
 - account for new, changed, or expired ratings

Numerical Example



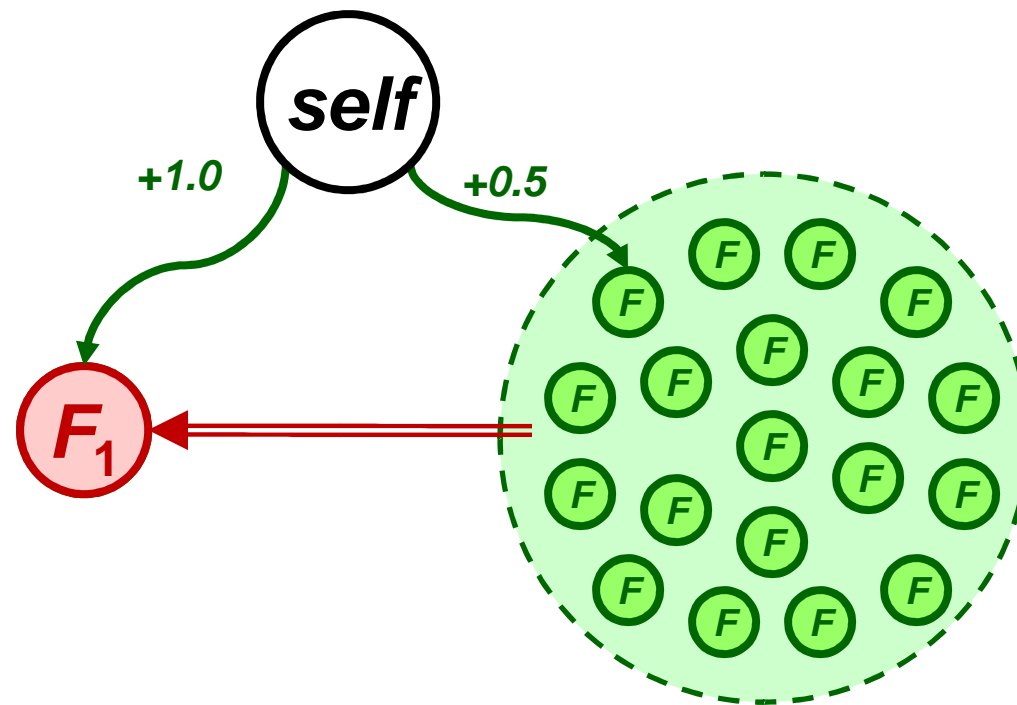
Corner Cases

- Dedicated adversaries to acquiring positive reputation and sharing it with malicious



Corner Cases cont'd

- Peers drastically overturning own ratings
 - consider it an intervention



GetReputation()

- Simply return R_j and $r_{i,j}$
 - very efficient
 - could easily add confidence/variation
- Various ways to convey the information
 - in mouse-over tooltip



- in player communication



- in group invites

Evaluation

- Implemented in optimized C++
- Emulated player population
 - using subset of the Slashdot Zoo

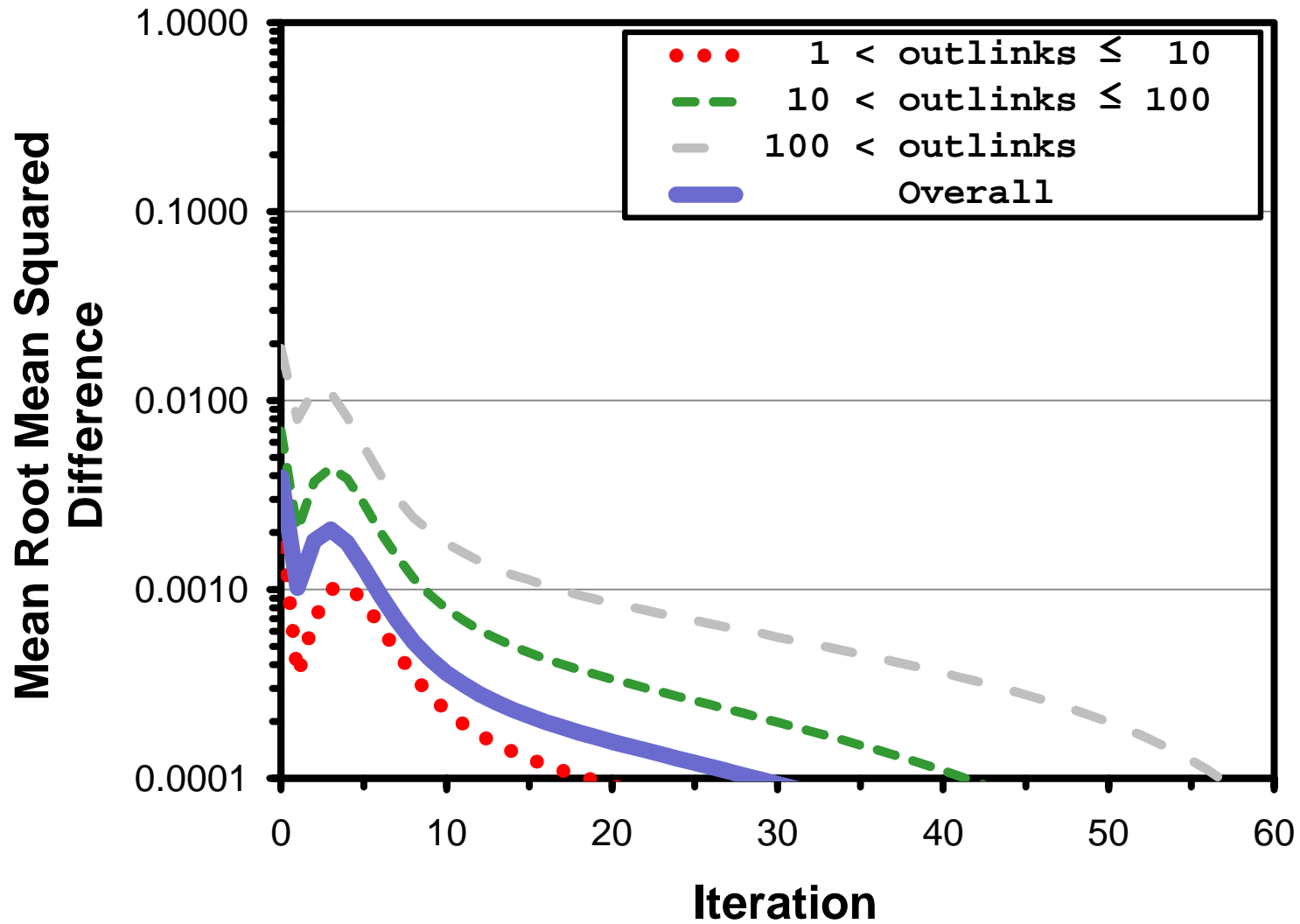
Characteristic	Value
Ratees	30,000
Raters	3,919
$1 \leq \text{ratings} < 10$	2,402
$10 \leq \text{ratings} < 100$	1,230
$100 \leq \text{ratings}$	287
Total Ratings	101,842
positive	76,101
negative	25,741
Mean Ratings	3.4
positive	2.5
negative	0.9
Sparseness	0.000113

Reputation Convergence

- Measure the change in reputations between `CalculateReputations()` iterations
 - Root Mean Squared Difference (RMSD)
 - Calculate the average RMSD across all peers with at least one rating

$$RMSD = \sqrt{\frac{\sum_{i \in Players} (R'_i - R_i)^2}{|Players|}}$$

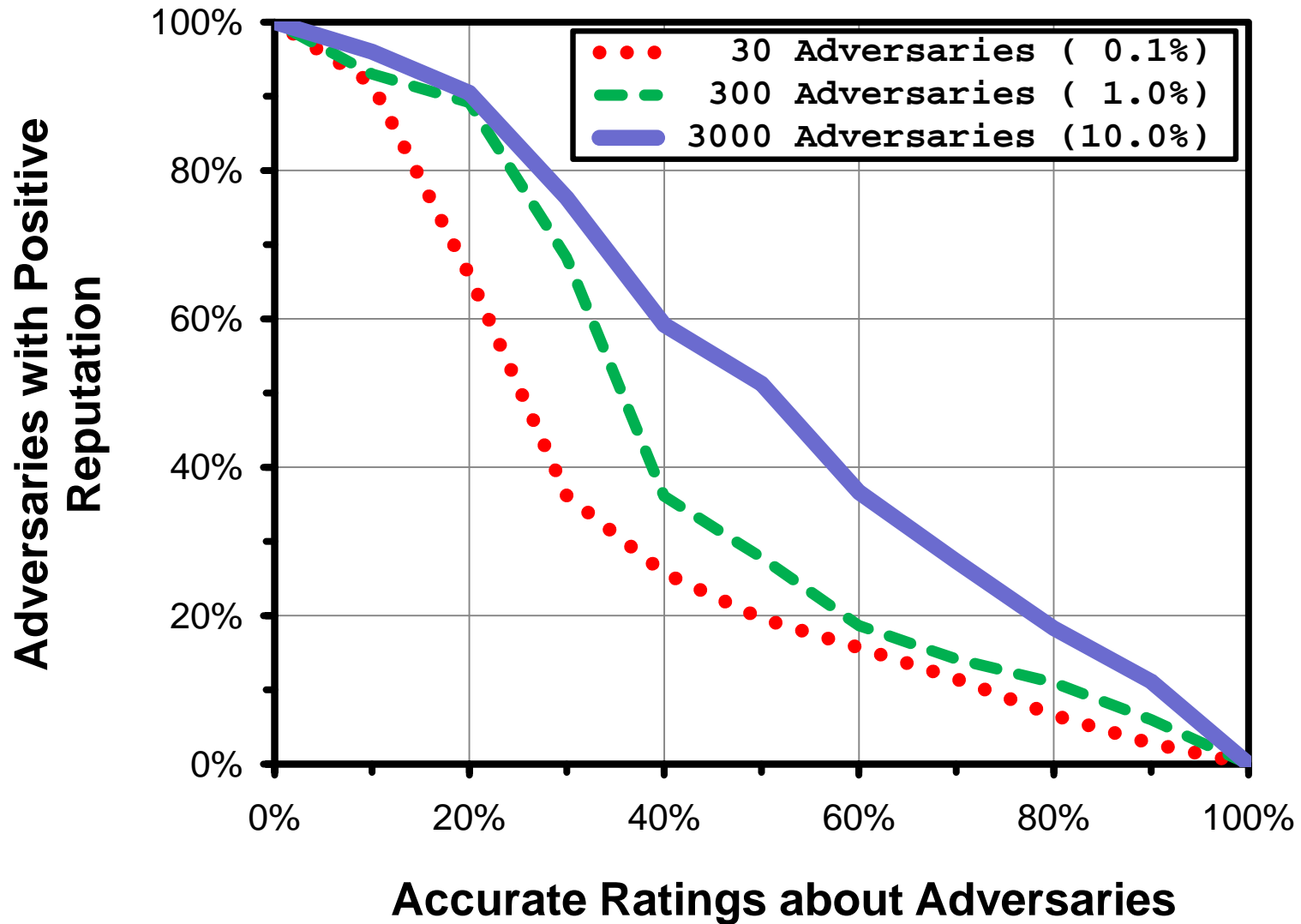
Reputation Convergence cont'd



Collusion Resistance

- Assume original graph is legitimate
- Add adversary population
 - fixed percent of original (0.1%, 1.0%, 10.0%)
 - fully connected with positive links
- Initially adversaries behave nicely to earn positive ratings from original peers
- Slowly start abusing trust (negating positive ratings)
- Calculate mean percentage of adversaries with positive reputation (as viewed by original peers)

Collusion Resistance cont'd



Observations

- Reputations converge relatively quickly
 - within tens of iterations
 - despite sudden ratings change
 - resistant to oscillation
- Resistant to collusion attacks
 - adversary discovery quickly corrects reputation
 - could hasten correction by weighting negative ratings heavier

Applications

- Better match-making functions
 - account for the possibility that certain players may or may not enjoy playing together
- Implement a recommender system on top
 - for systems that offer multiple game titles
 - “*people you enjoyed playing game X with also play game Y*”

Limitations

- Server doesn't observe this data yet
- Requires unique player identity
 - can be changed (but only to another unique identity)
- Player behavior may change suddenly
 - sell accounts with good reputation
 - raters will have to adjust their own ratings
- Single metric doesn't distinguish rating on component factors (e.g., game skill, personality, etc.)
 - scaled PlayerRating linearly in storage, computation, and communication

Conclusions

- Players misbehave
 - whether premeditated or not
 - hard for game developers to adequately police
- **PlayerRating** is specifically designed to allow good players to congregate
 - facilitates incremental deployment
 - encourages accurate participation
 - resists abuse
 - imposes minimal overhead

Thanks

Extra Slides

CalculateReputations()

```
1:  $R', w \leftarrow \emptyset$ 
2: for all  $i \in \text{Players}$  do
3:    $w_\Delta \leftarrow \text{Influence}(R_i)$ 
4:   if  $w_\Delta \leq 0$  then skip  $i$ 
5:   for all  $r_{i,j} \neq 0.0$  and  $j \neq \text{self}$  do
6:      $R'_\Delta \leftarrow (r_{i,j} \times R_i \times \text{Decay}(TTL_{i,j})) - R'_j$ 
7:      $w_j \leftarrow w_j + w_\Delta$ 
8:      $R'_j \leftarrow R'_j + (R'_\Delta \times w_\Delta / w_j)$ 
9:   end for
10: end for
11:  $R \leftarrow R'$ 
```

weighted average calculation

only blocking action