Scaling DDS

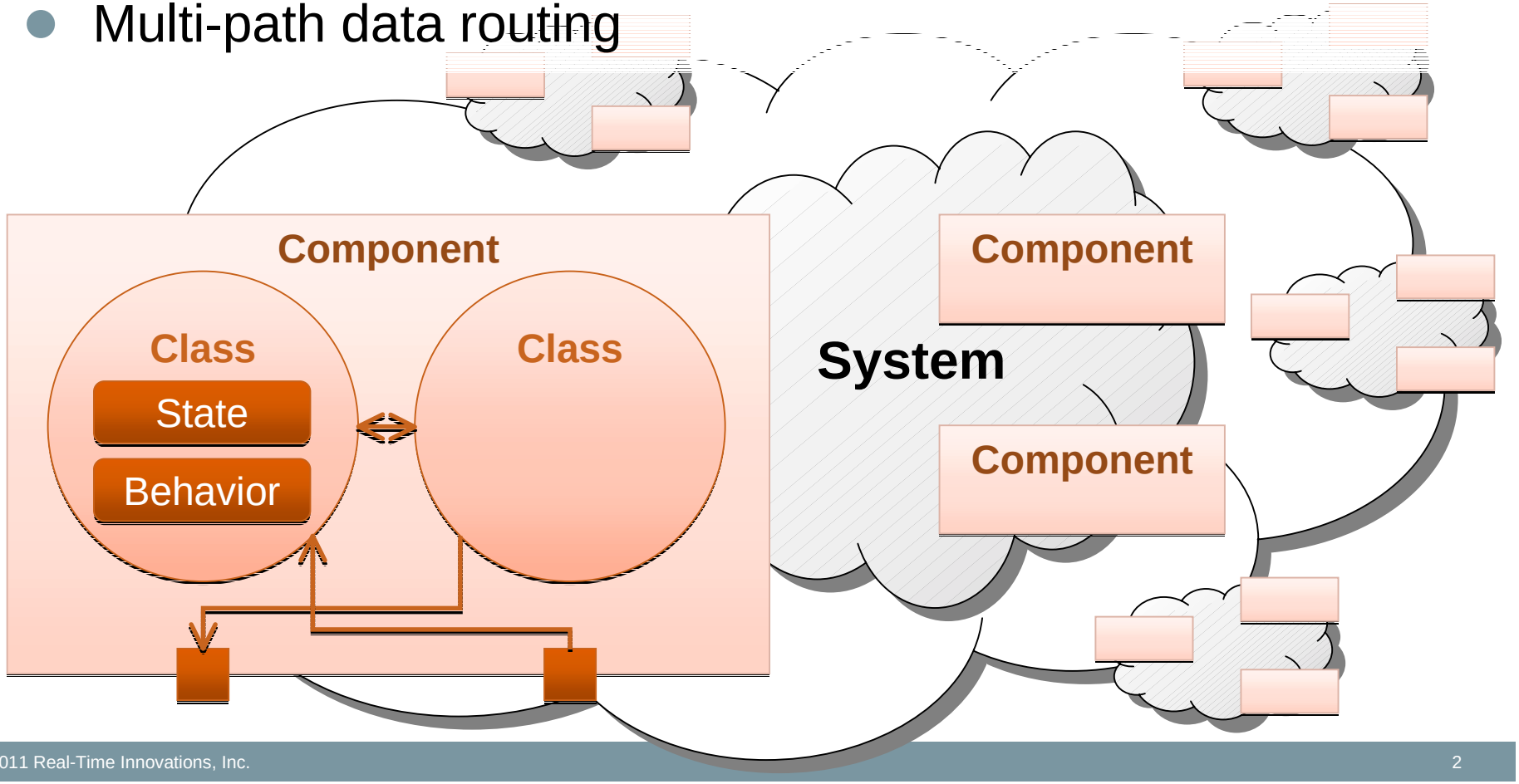
to Millions of Computers and Devices

Rick Warren, Director of Technology Solutions
rick.warren@rti.com

Last Year in Review: "Large-Scale System Integration with DDS"

Recommended scaling approach:

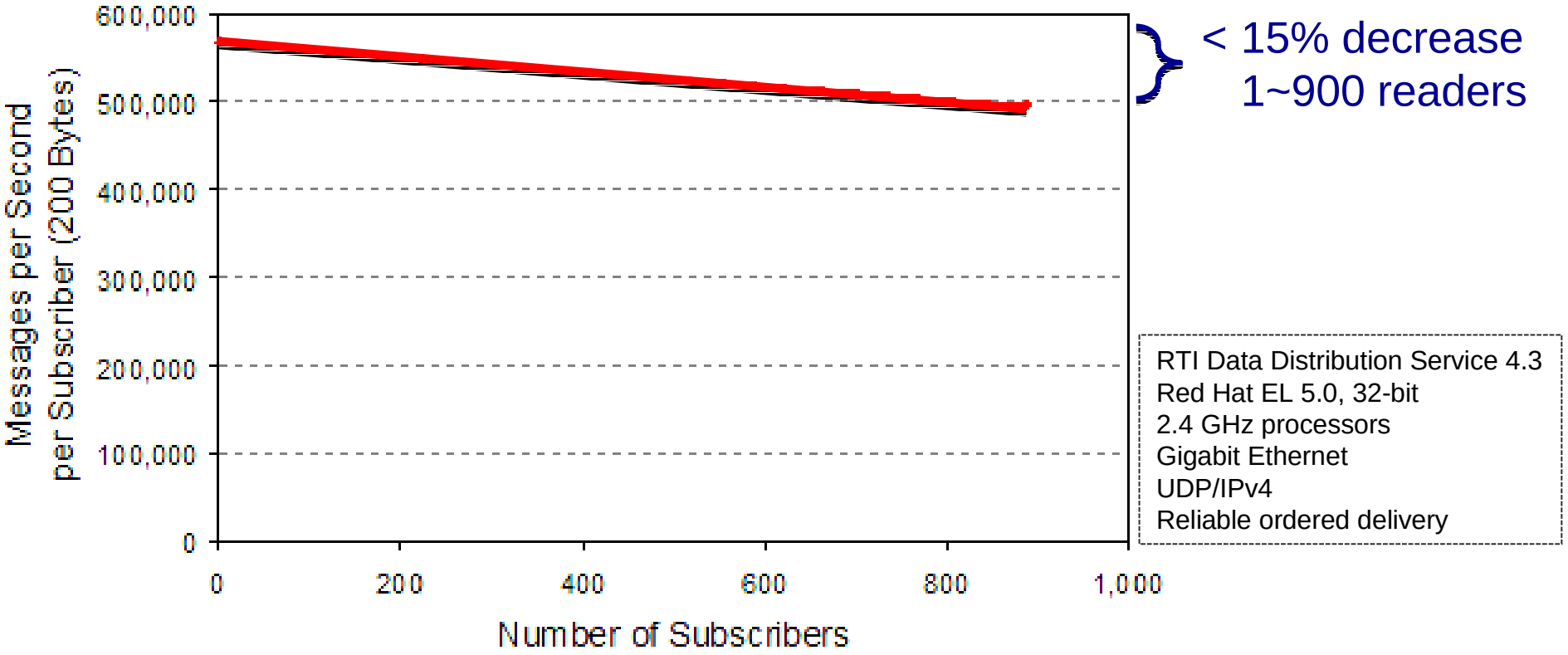
- Hierarchical composition of subsystems
- Multi-path data routing



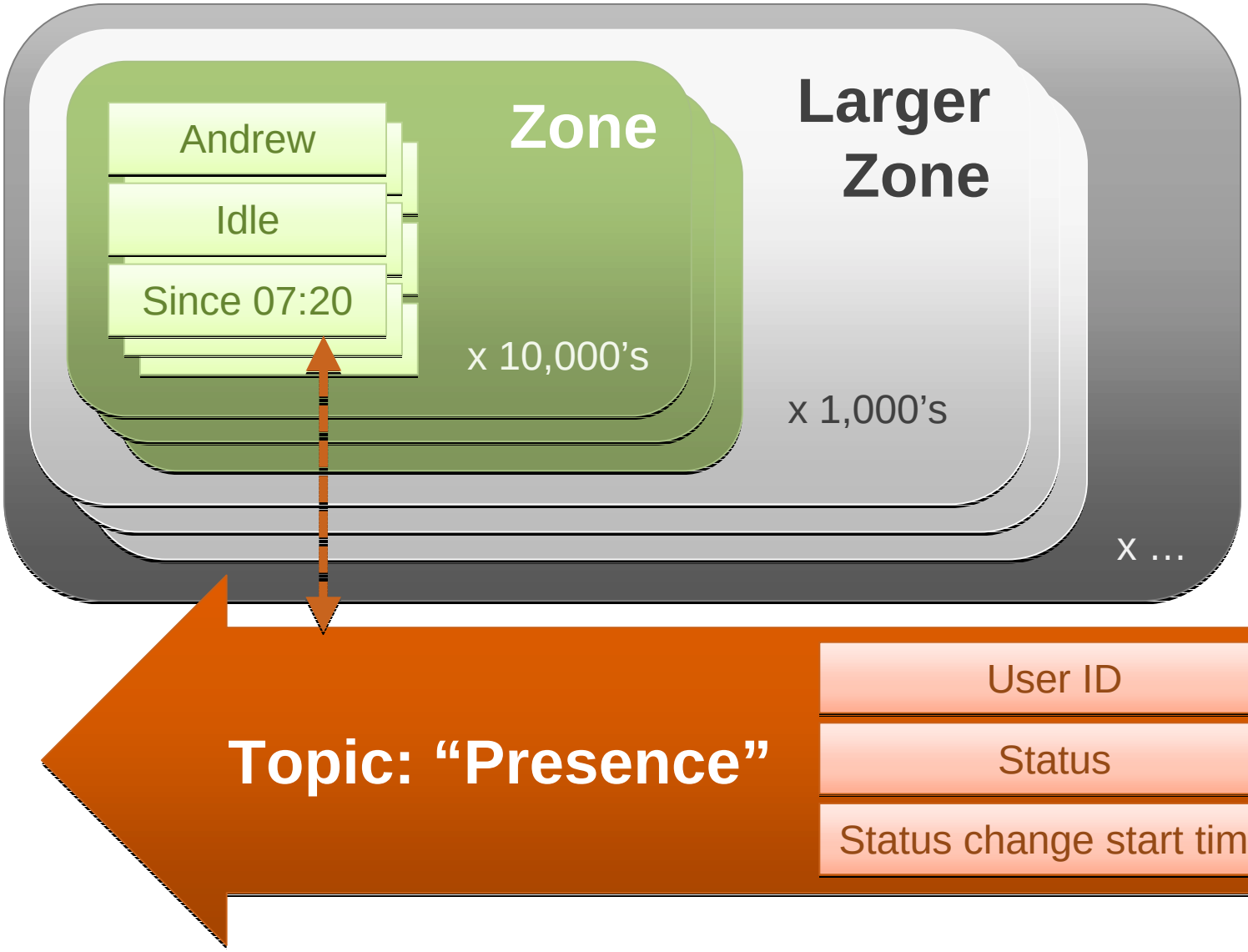
Last Year in Review: "Large-Scale System Integration with DDS"

Proven within a fully connected subsystem:

- DDS-RTPS Simple Discovery of nearly 2,000 participants
- Large fan-out with little performance degradation



This Year: Example Scenario



- Many users
- Each an instance of a common topic
- Subscribe to your friends

Problem Statement

- Network designed hierarchically
  - Peers subscribe to their own data
  - Peers may subscribe on behalf of other peers
- Large numbers of publishers and subscribers, distributed geographically
- Volume of data in total is very large
  - …compared to the volume of data of interest
  - …at any one location
- **Must scale no worse than O(n)**, where n == # parties producing and/or consuming a certain set of data
  - …in terms of bandwidth requirements
  - …in terms of state storage requirements
  - …in terms of processing time

Problem Statement—Translation for DDS

- Routed network
  - …at network / transport levels (Layer 3, Layer 4)
  - …at data level
- Unicast communication
  - No routable multicast
  - Probably crossing at least one firewall
- Instance-level routing and filtering critical, in addition to topic-level
- Must consider distribution of application data + discovery data

Products available and interoperable based on existing standards.

DDS-RTPS TCP mapping available. Standardization kicking off. Not the subject of this talk.

Future R&D needed based on principled extensions to existing stds.

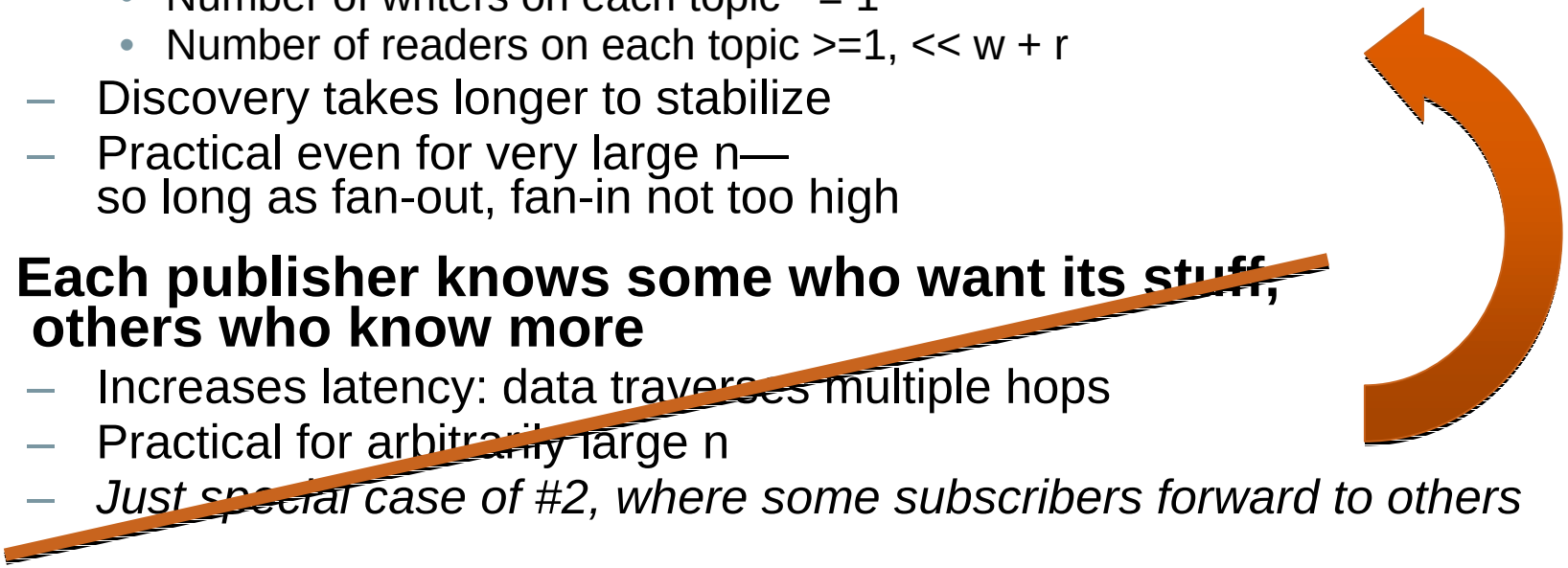
Data Distribution

Algorithmic Scalability

- **Sending time is unavoidably linear**
  in number of destinations
  - Unicast delivery means must `send()` to each
  - *Implication*: Avoid other factors > O(log n)—preferably O(1)
    - e.g. Don't perform a O(n) calculation with each send
    - Keep *n* small! Don't send stuff to destinations that don't care about it.
- **Discovery state growth is unavoidably linear [O(n)]**
  in total system size
  - Data must be maintained durably for each application, publication, subscription—at least once
  - (Just like an IM or email system must keep a record of each user)
  - *Implication*: Avoid other factors > O(log n)—preferably O(1)
    - e.g. Don't store information about everything everywhere: $O(n^2)$
    - Keep *n* small! Don't store or transmit things you don't care about.

Distributed Data Routing Approaches

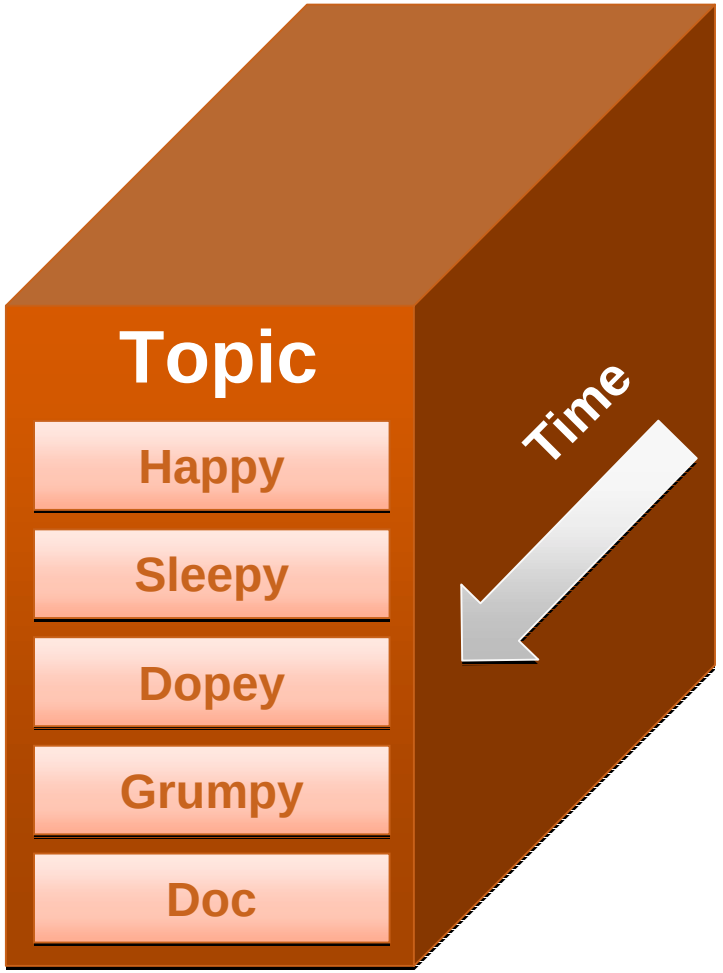
1. **Everyone knows who wants anything**
   - State scales $O((w+r)^2)$
     - w = number of data writers
     - r = number of data readers
   - Discovery stabilizes very quickly for small n, changes are also quick
   - Impractical for large n
2. **Each publisher knows who wants its stuff**
   - State scales ~O(kw + r)
     - Number of writers on each topic ~= 1
     - Number of readers on each topic >=1, << w + r
   - Discovery takes longer to stabilize
   - Practical even for very large n—
so long as fan-out, fan-in not too high
3. **Each publisher knows some who want its stuff, others who know more**
   - Increases latency: data traverses multiple hops
   - Practical for arbitrarily large n
   - *Just special case of #2, where some subscribers forward to others*



Distributed Data Routing Approaches

Example scenario: 1 topic, very many instances, few writers/readers for each

- **Everyone gets all instances** (…on that topic)
  - Network traffic scales ~O(wr)—*quadratic*
  - Aggregate subscriber filtering CPU scales ~O(wr)
- **Each subscriber gets only instances of interest**
  - Network traffic scales ~O(w)—*linear, assuming few readers per writer*
  - Aggregate subscriber filtering CPU scales <= ~O(r)—depending on whether readers have additional filtering to do
  - *Needed*: way to specify instance-level publications, subscriptions
  - *Needed*: filter implementation scalable in space, time



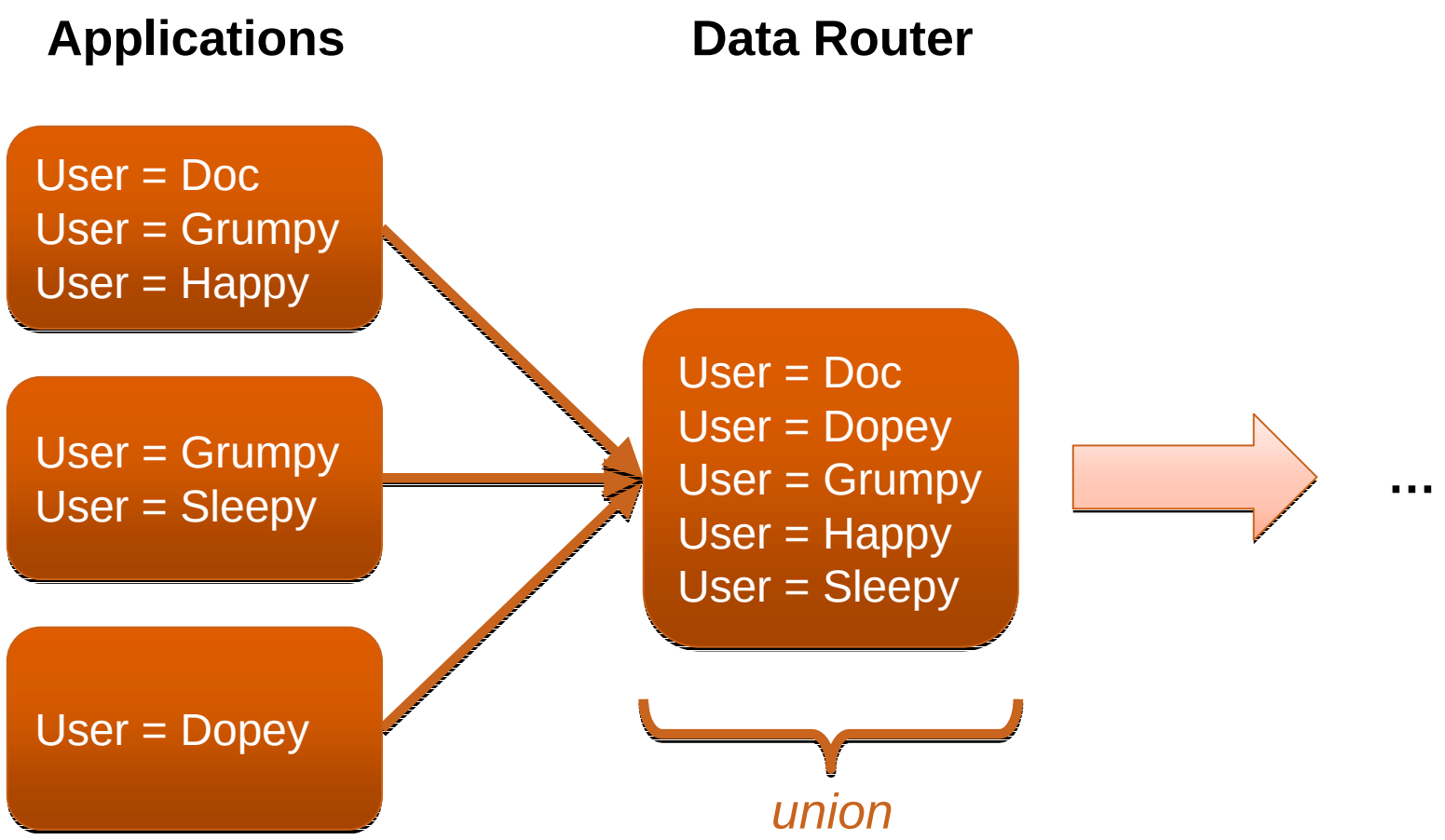
DDS Enhancements

Expressing Instance-Level Interest

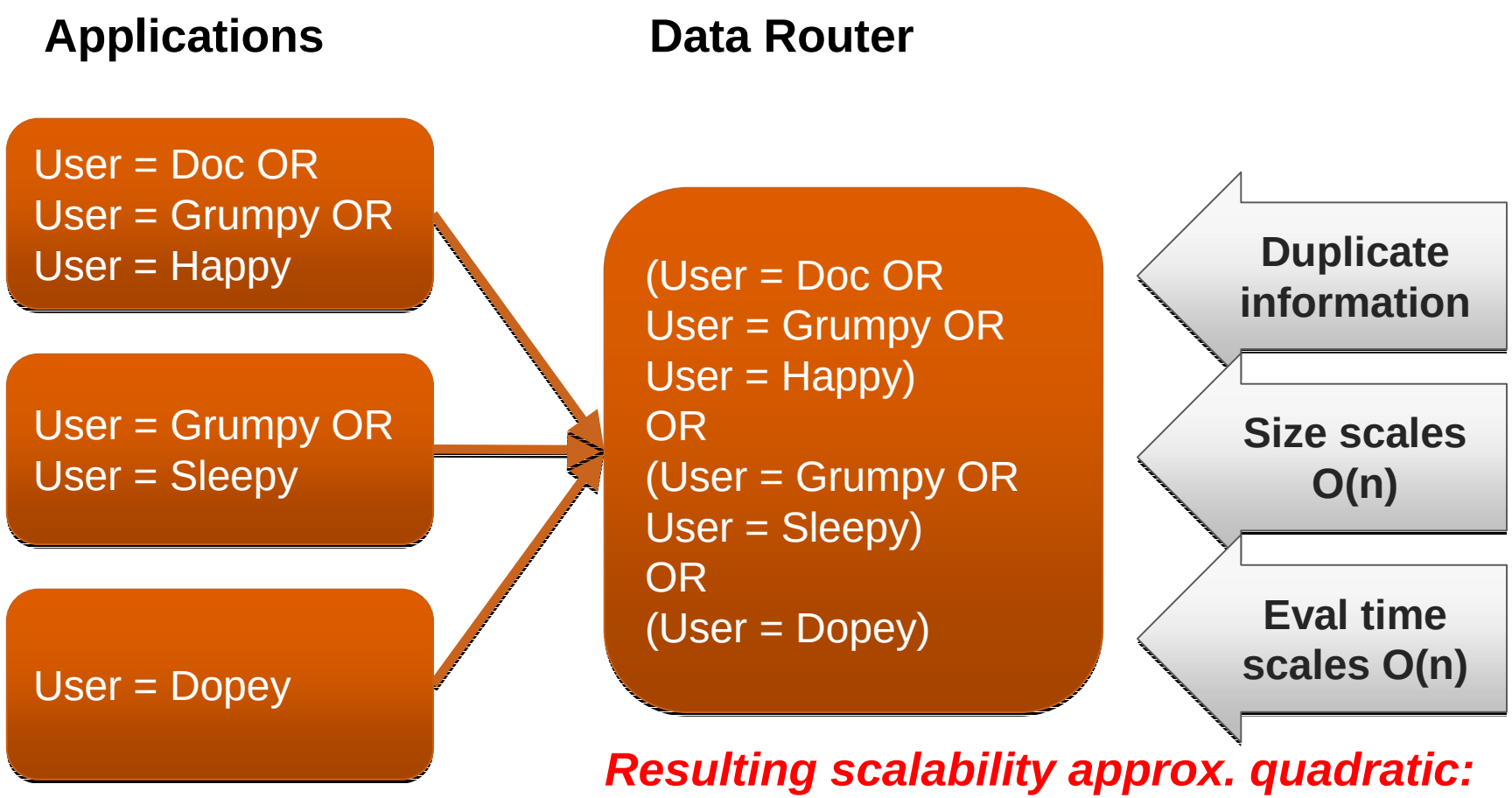
- **Recommended enhancement**: discoverable instance-level interest/intention on *both* pub and sub sides
  - For efficient management by middleware implementations
  - *Example*: Discovery data: don't propagate pub data to non-matching subs

- ***Publishing***
  - *Today*: Infer from writer instance registration (`register()` or `write()`)
  - *Potential future*: Pre-declared, e.g. through QoS policy
- ***Subscribing***
  - *Today*: Evaluate based on reader content-based filter
  - *Potential future*: Explicit instance expression, intersected with content filters

Efficient Instance Filters: Use Case



Efficient Instance Filters? Today's SQL Filters



Problem

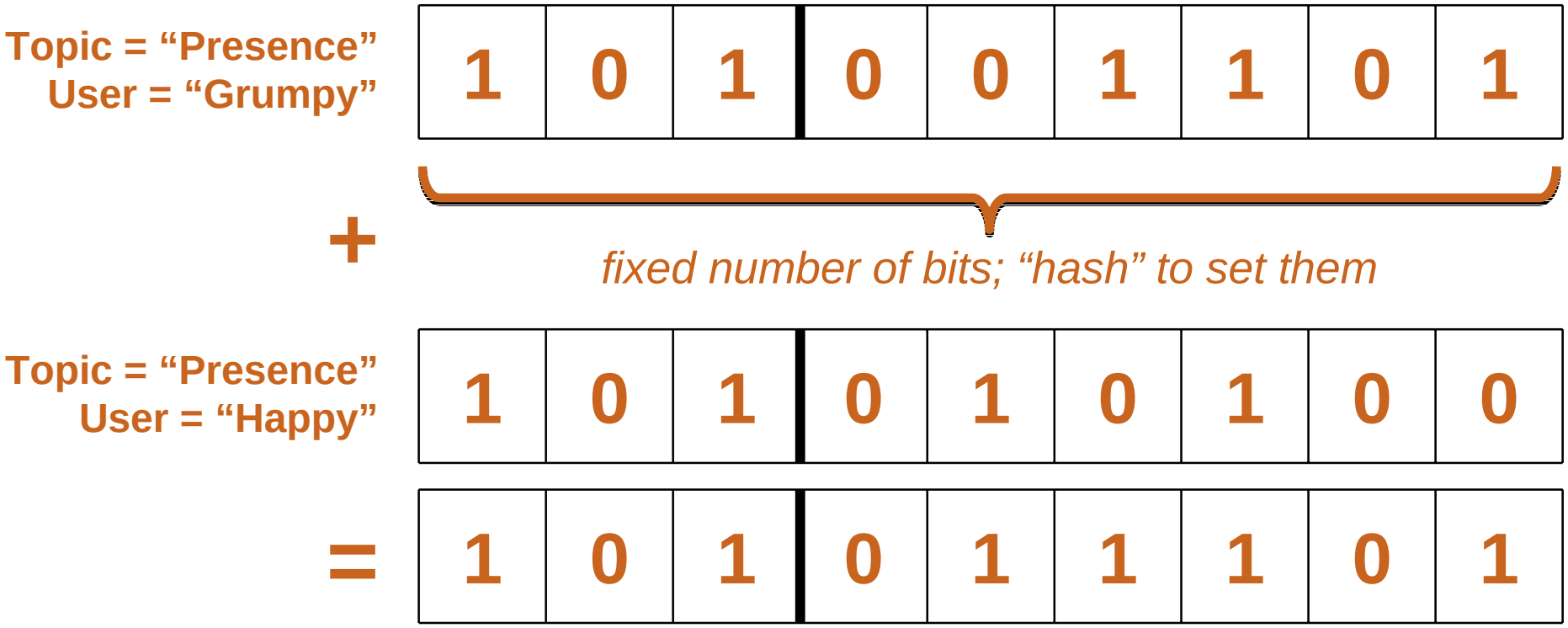
Resulting scalability approx. quadratic:

Each send from "outside" router pays price proportional to system size "inside".

Efficient Instance Filters: Bloom Filters

- **Designed for filtering based on membership in a set**
  - Total potential number of elements in set may be large—even infinite
  - Evaluation may have false positives, but never false negatives
  - *Well suited to representing instances in a topic*
- **Compact representation**
  - Filter has fixed size, regardless of number of elements of interest
  - Larger size => lower probability of false positives
  - *Well suited to interest aggregation*
- **Fast updates and evaluations**
  - Adding to filter: O(1)
  - Evaluating membership in filter: O(1)
  - Removing from filter requires recalculation: O(n)
  - *Well suited to high performance with changing interest*

Efficient Instance Filters: Bloom Filters



- Union to add to filter; intersect bits to test membership
- **Filter size can be easily modeled**
- **Filter maintenance, evaluation costs are negligible**
- ***Open challenge*: testing set overlap requires very sparse patterns**

Per-Instance Late Joiners

- **DDS-RTPS implements reliability at topic level**
  - Per-*instance* sequence numbers, heartbeats, state maintenance too expensive for large numbers of instances

- **Challenge**: how to be "late joiner" at instance level?
  - *For example*:
    - There are 10 million people (instances).
    - I cared about 20 of them. Now I care about 21.
    - How do I negatively acknowledge just the one instance?
  - *Implication*: subscribers may need to cache "uninteresting" information *in case* it becomes interesting later
    - Simple Endpoint Discovery is example of this

Per-Instance Late Joiners

- **DDS-RTPS implements reliability at topic level**
  - Per-*instance* sequence numbers, heartbeats, state maintenance too expensive for large numbers of instances

- **Response**: Several possible protocol improvements
  - Maintain per-instance DDS-RTPS sessions anyway
    - Number of instances per writer/reader is small in this use case
  - …or maintain exactly 2 DDS-RTPS sessions: live data + "late joiner" instances
  - …or introduce new instance-level protocol messages
    - Supported by necessary state

Discovery

Distributing Discovery Data

- **Needed for discovery**:
  - Identities of interested parties
  - Data of interest—topics, instances, filters
  - Conditions of interest—reliability, durability, history, other QoS
  - Where to send data of interest
- ***Just data***—can be distributed over DDS like any other
  - Principle behind DDS-RTPS "built-in topics"
    - …for participants
    - …for publications and subscriptions
    - …for topics
- **One more step: bootstrapping**
  - Discovery service provides topics to describe the system …but how to find that discovery service?
  - May be statically configured statically or dynamically-but-optimistically

What is a DDS "Discovery Service"?

- **Provides topics to tell you about DDS objects** (or some subset):
  - Domain participants
  - Topics
  - Publications
  - Subscriptions
- **Must be bootstrapped by static configuration**
  - Fully static configuration of which services to use—OR
  - Static configuration of *potential* services (locators) + dynamic discovery of *actual* services
    - Dynamic information must be propagated optimistically, best effort
    - Scope of this presentation

Today: DDS-RTPS "Simple Discovery"

Two parts:

1. Participant Discovery

- Combines discovery bootstrapping with discovery of 1 participant
- Advertizes (built-in) topics for describing arbitrary (system-defined) topics, pubs, and subs
- *Any Participant Built-in Topic Data sample can represent a discovery service*

Recommended additions:

- *For scalable filtering, partitioning of discovery data*: Include built-in topic data for other built-in endpoints
- *For limiting endpoint discovery*: Bloom filter of aggregated interest across all topics
- *For aggregation of participant information*: New reliable built-in topic for propagating data for other participants ("participant proxy")

Today: DDS-RTPS "Simple Discovery"

Two parts:

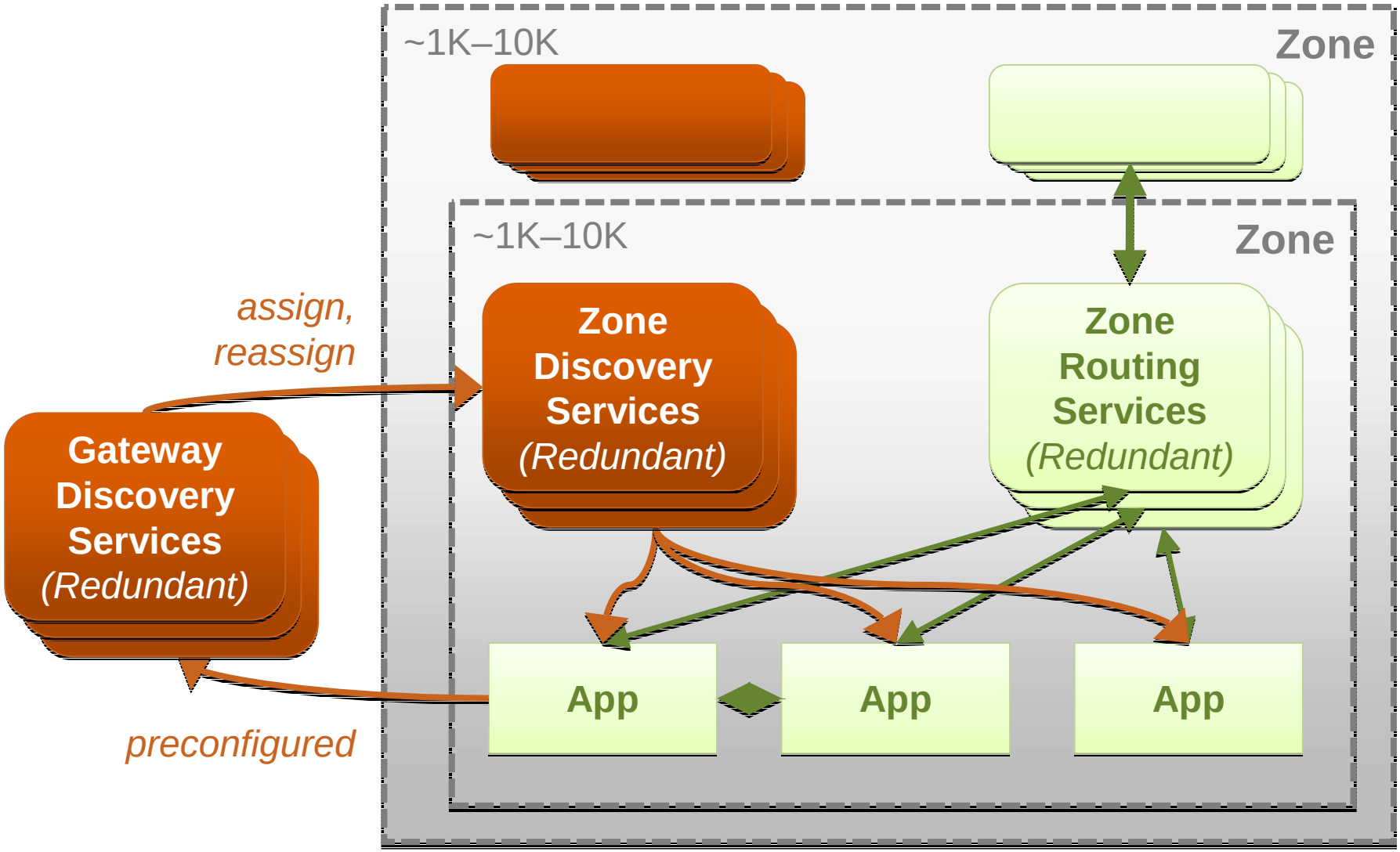
2. Endpoint Discovery

- Regular DDS topics for describing topics, publications, subscriptions
  - Reliable, durable—cache of most recent state
- Assumes known discovery service—provided by Participant Discovery
  - Service may propagate "its own" endpoints or forward endpoints of others using existing DDS mechanisms

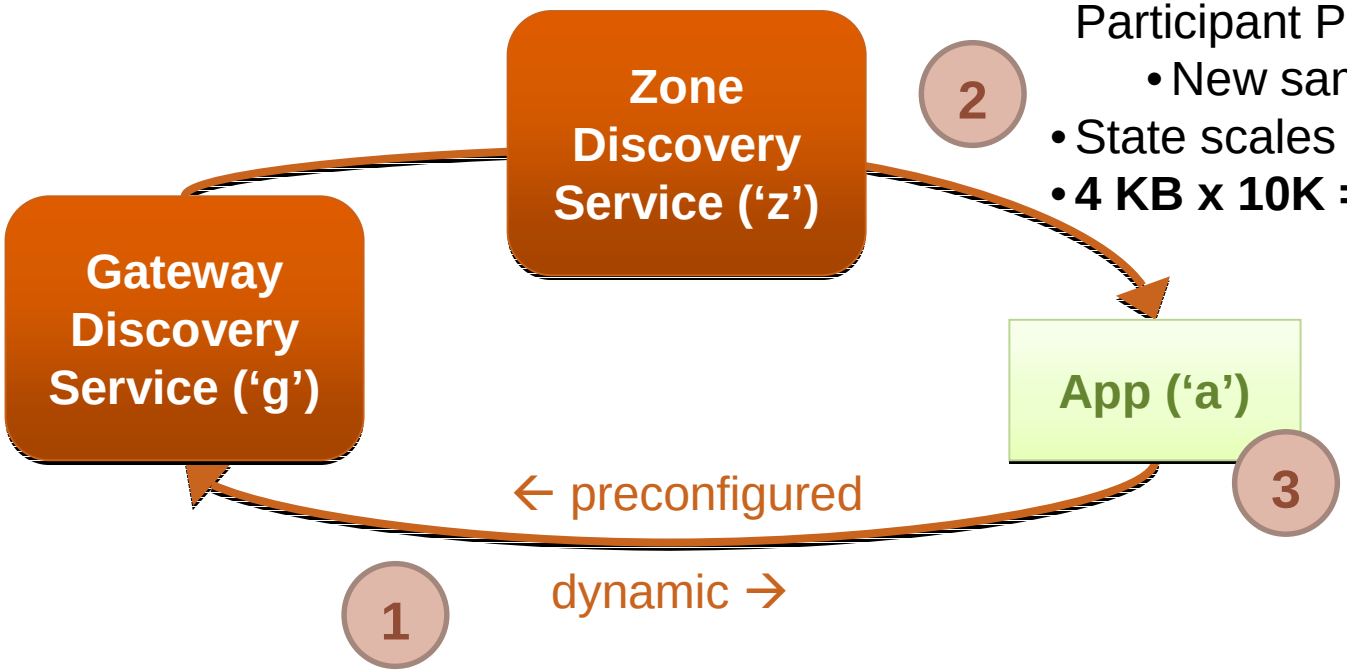
Recommended additions:

- *To limit discovery of unmatched endpoints*: Bloom filters to express instance interest
- (Proposed "participant proxy" topic would fit into this model)

Putting It Together: Scalable Communication



Putting It Together: Participant Discovery

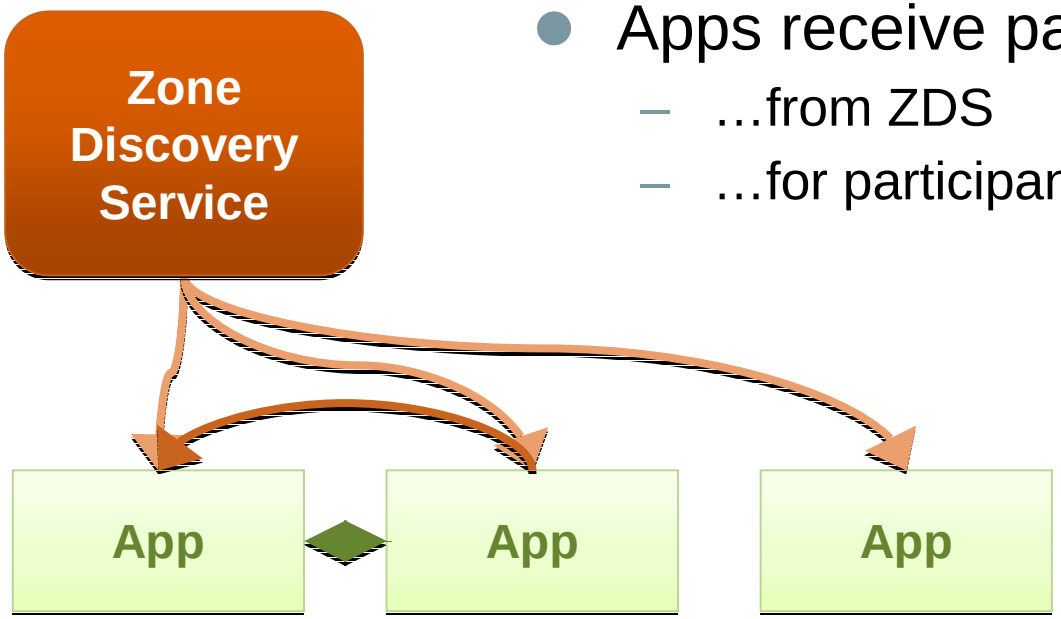


- GDS forwards ZDS over proposed Participant Proxy built-in topic
  - New samples only on change
- State scales $O(a_z)$ ('a' in this zone)
- **4 KB x 10K = 40 MB**

- Use existing Participant built-in topic
  - …with very short-lived connections
- State scales O(z) + O(a) ('a' across all zones)
  - Expected z = 1K; a = 10M but expire quickly
- **4 KB x (1K + 0.01 x 10M) = 400 MB**

- App only talks to ZDS Participant Proxy
- ZDS announces participants matching app's filter; app does endpoint discovery with that participant directly
- State scales $O(z_z)$ + O(p)
  - z_z == 2 or 3

Putting It Together: Endpoint Discovery

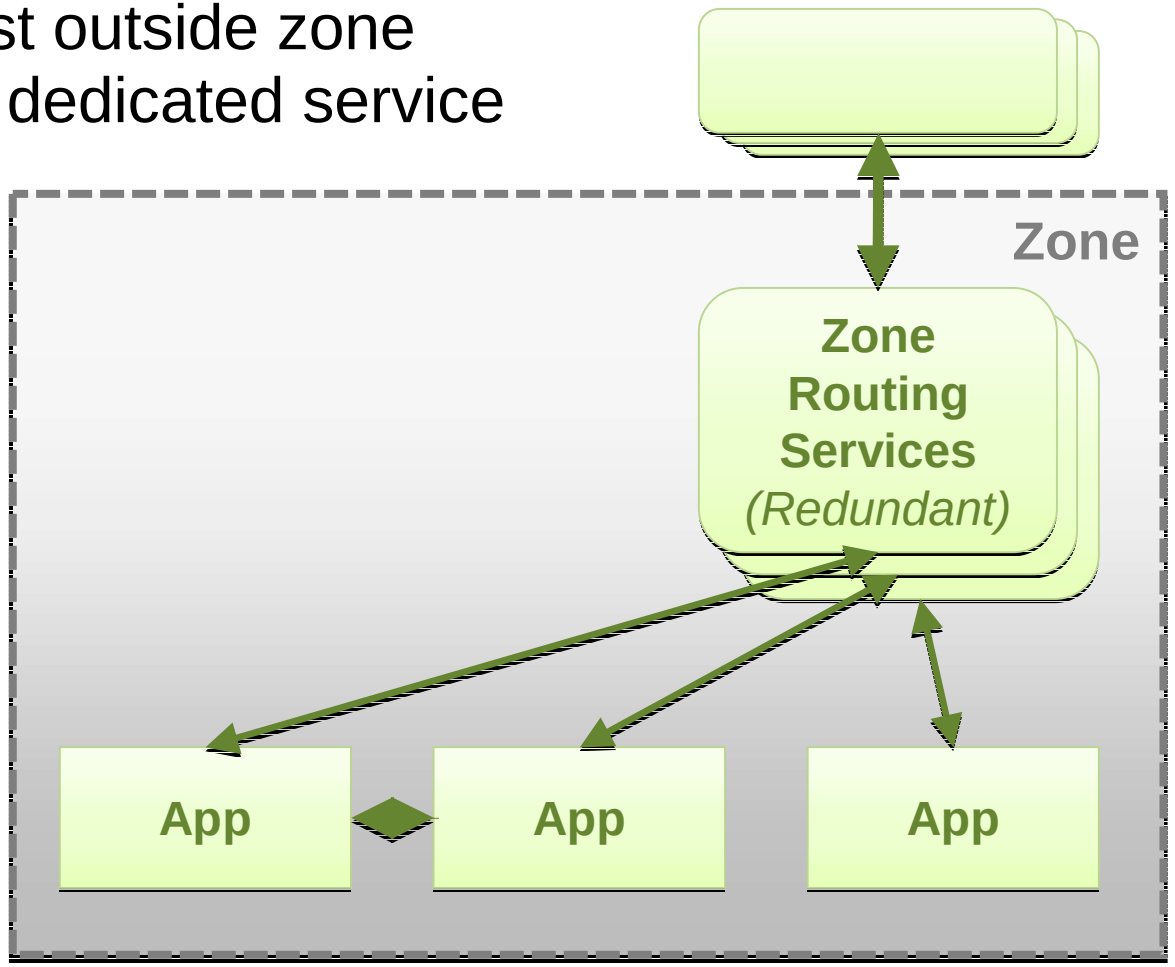


- Apps receive participant proxy records
  - …from ZDS
  - …for participants with matching endpoints

- Apps receive matching endpoints
  - …directly from apps that have them
  - …based on filters attached to built-in readers
  - *Do not discover non-matching endpoints*
  - *Treated as late joiners for new matches when endpoints change*

Putting It Together: Data Distribution

- Traffic of interest outside zone routed there by dedicated service
- Traffic within zone flows directly, peer to peer



Conclusion

- **Last year's talk: demonstration of current scalability**
  - Thousands of collaborating applications
  - Based on:
    - Federated and/or composed DDS domains
    - Multi-path data routing

- **Today: roadmap for scaling DDS out further**
  - Millions of collaborating applications
  - Based on:
    - Managing discovery over the WAN
    - Improving efficiency of data filtering, routing for all topics—discovery also benefits
  - *Based on incremental extensions to existing standard technology*

Thank You