How to build large scale distributed systems on self-organization?

Márk Jelasity

University of Szeged and
Hungarian Academy of Sciences

Motivation

- Massively large scale distributed systems are now common
  - clouds, (desktop) Grid, P2P
- We want them to be cheap, available, reliable, robust
- bottom-up self-organization (emergence) is a promising mindset to try here (cheap, robust, etc)
- But how to build systems out of it?

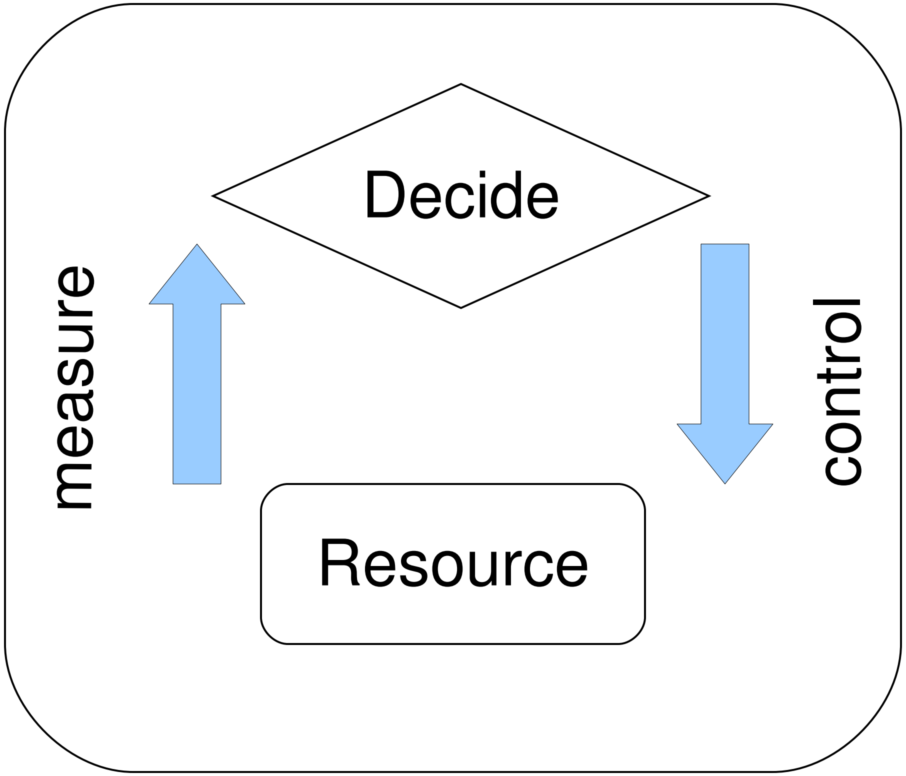
Outline

- What is a self-organizing service? Why self-organization?
- Examples of such services
  - gossip based overlays and monitoring
- How to combine such services?
- An example application
  - heuristic global optimization
- Toward a general purpose architecture

non-self-organizing self-management

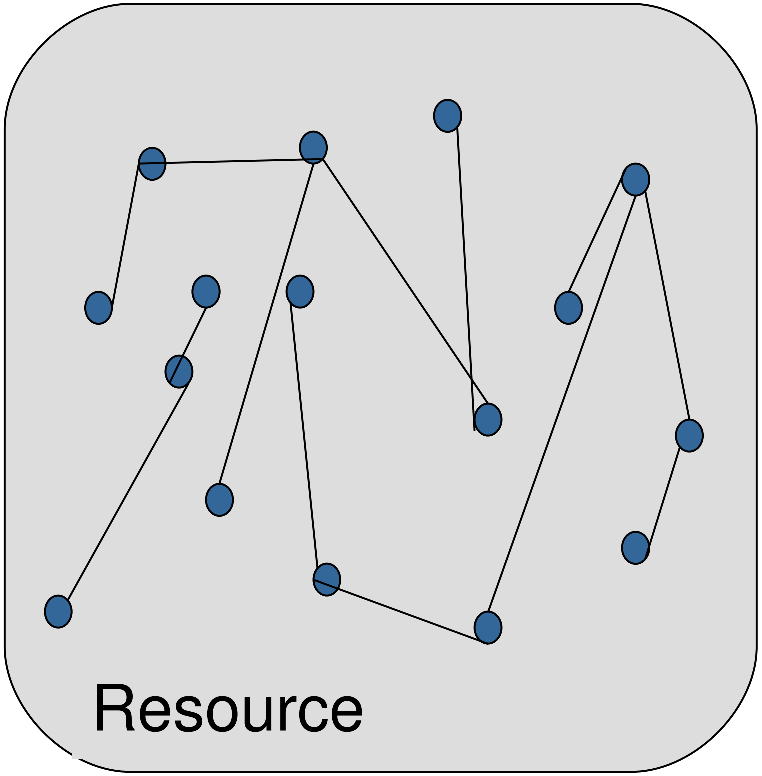
``The autonomic computing architecture starts from the premise that implementing self-managing attributes involves an intelligent control loop''

An architectural blueprint for autonomic computing, IBM

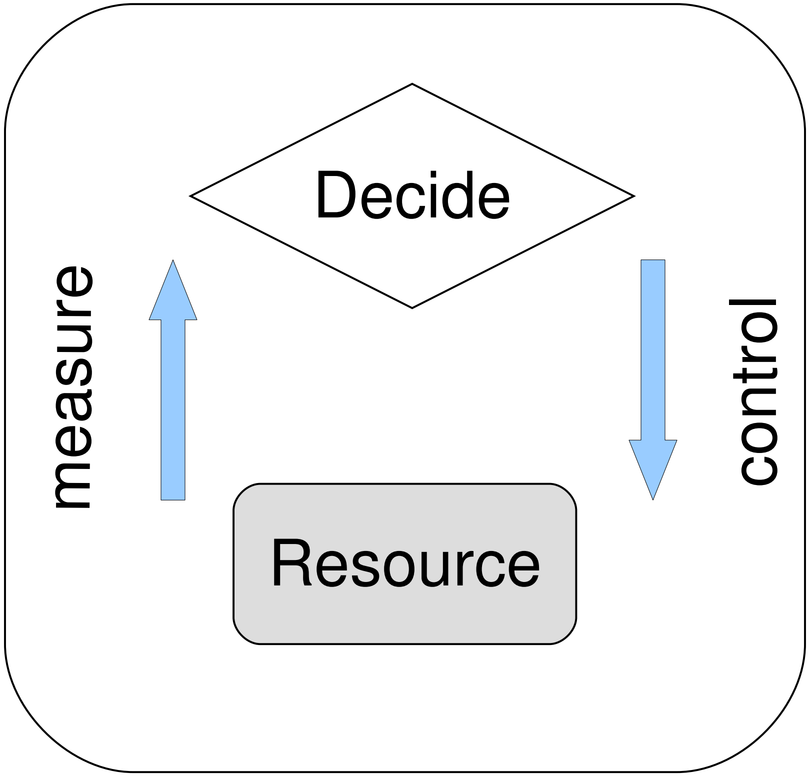


self-managing resource

Self-organization



self-organizing resource



control loop

comparison of bottom-up and top-down self-management

- emergence
- extreme simplicity
- no explicit knowledge (representation)
- no explicit decisions
- no separation of manager/managed at any level
- low predictability (?)

- ML, AI (even GOFAI):
- symbolic approach
- high (extreme?) complexity
- knowledge based, often explicit conceptual, rule based knowledge representation (policy)
- separation of managed entity and manager (homunculus)
- predictability (?)

Some comments on self-organization

Not a universal solution, probably appropriate in very large scale, highly dynamic, highly distributed systems

- Potentially robust and scalable
- Much easier to implement
- Therefore computer architectures do not become orders of magnitude more complex (like in autonomic computing)
- Potentially more efficient and effective
- Has its open problems too:
  - predictability and controllability
  - new design philosophy: not a gradual transition

The trust problem

End users and administrators can hardly trust emergent systems because

- often there is no hard guarantee they do what they supposed to
- even when there is scientifically established guarantee, there is no sense of understanding: human cognitive gap between microscopic and macroscopic behavior
- there is no sense of control (due to cognitive gap): what action is necessary to achieve X?

Combining components

- Let us make the grassroots protocols managable through modularity
  - simple components (building blocks, services) for a specific simple function
  - due to simplicity they can be understood scientifically with a larger chance of success
  - they can be thoroughly understood, described and explained to non-researchers
  - they can be combined (now in a non-emergent manner) to form new, more complex functions keeping the benefits of simplicity, robustness and scalability

A Gossip Skeleton

- Originally for information dissemination in a very simple but efficient and reliable way
- Later the gossip approach has been generalized resulting in many local probabilistic and periodic protocols
- we will introduce a simple common skeleton and look at
  - information dissemination
  - topology construction
  - aggregation

A Gossip Skeleton

- the push-pull model is sown
- the active thread initiates communication (push) and receives peer state (pull)
- the passive thread mirrors this behavior

do once in each T time units at a random time

    p = selectPeer()
    send state to p
    receive $state_p$ from p
    state = update($state_p$)

active thread

do forever

    receive $state_p$ from p
    send state to p
    state = update($state_p$)

passive thread

Rumor mongering as an instance

- state: set of active updates
- selectPeer: a random peer from the network
  - very important component, we get back to this soon
- update: add the received updates to the local set of updates
- propagation of one given update can be limited (max k times or with some probability, as we have seen, etc)

Peer Sampling

- A key method is selectPeer in all gossip protocols (influences performance and reliability)
- In earliest works all nodes had a global view to select a random peer from
  - scalability and dynamism problems

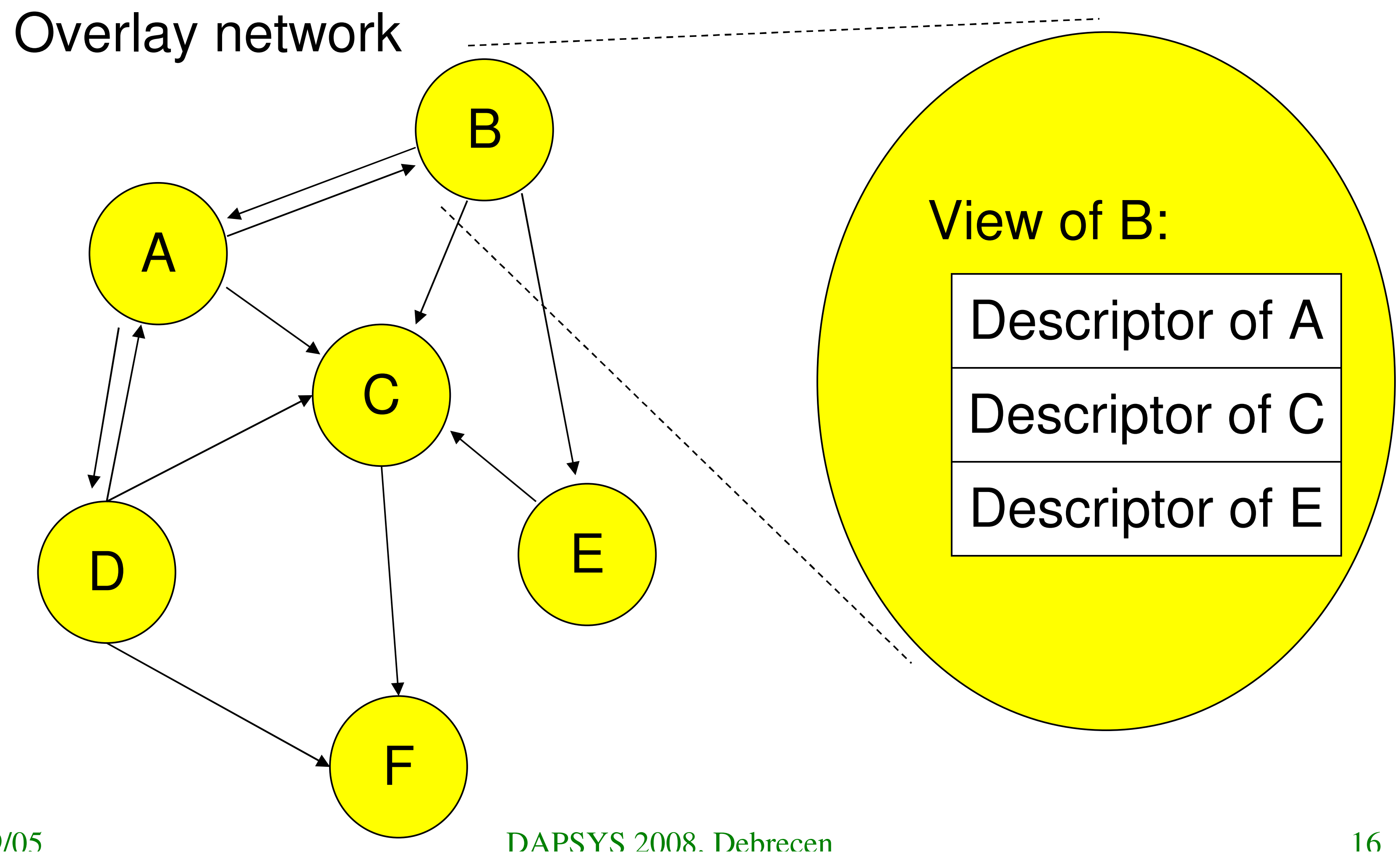
Gossip based peer sampling

- basic idea: random peer samples are provided by a gossip algorithm: the peer sampling service
- The peer sampling service uses itself as peer sampling service (bootstrapping)
  - no need for fixed (external) network
- state: a set of random overlay links to peers
- selectPeer: select a peer from the known set of random peers
- update: for example, keep a random subset of the union of the received and the old link set

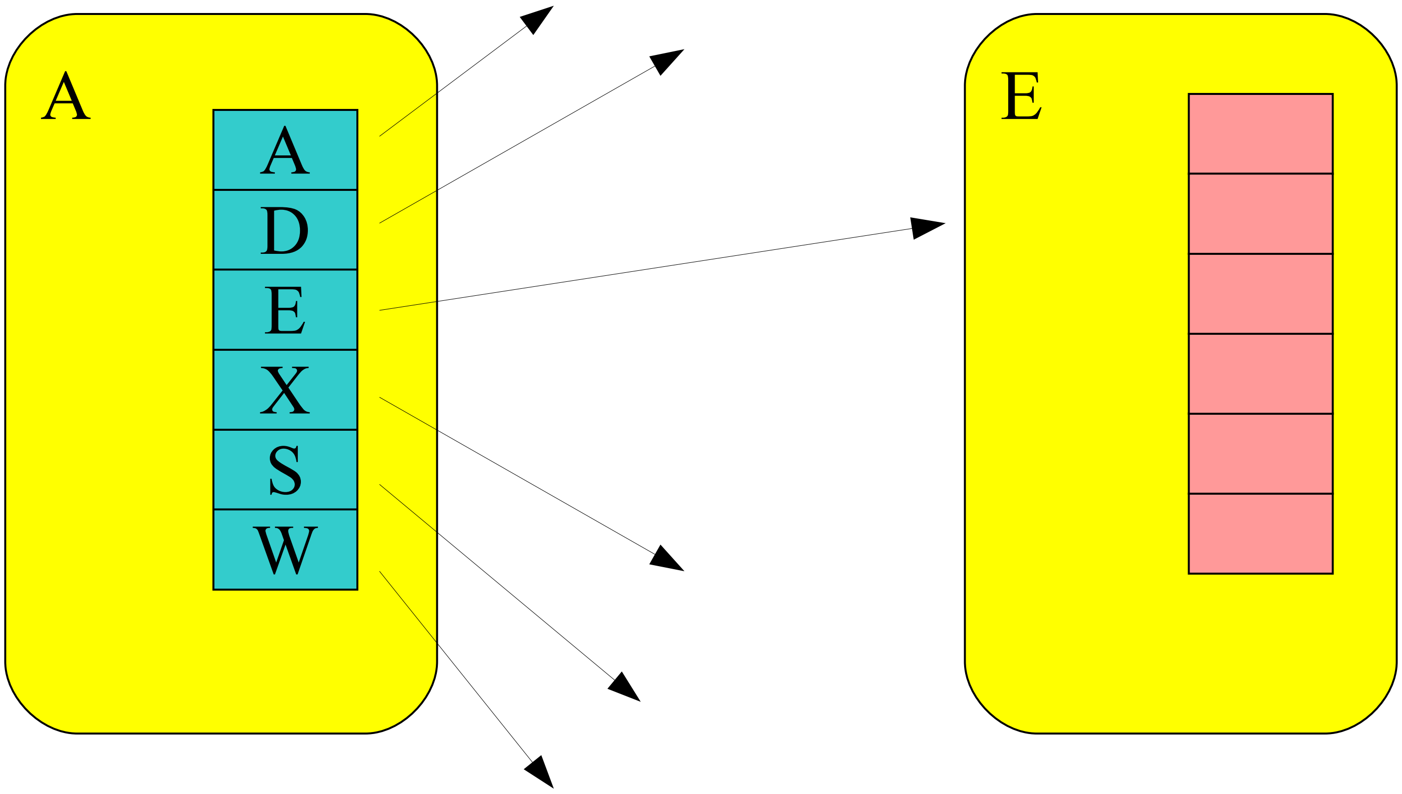
Overlay Networks

- Most problems boil down to building and maintaining overlay networks: the protocols then are simple
  - peer sampling: random overlay
  - bootstrapping: arbitrary overlay
- overlay networks
  - Nodes are computing devices connected to a computer network
  - Neighbours are defined by the “knows-about” relation (NOT physical neighbors in the network).
    - **Can be easily adapted (unlike physical networks) by simply exchanging information**

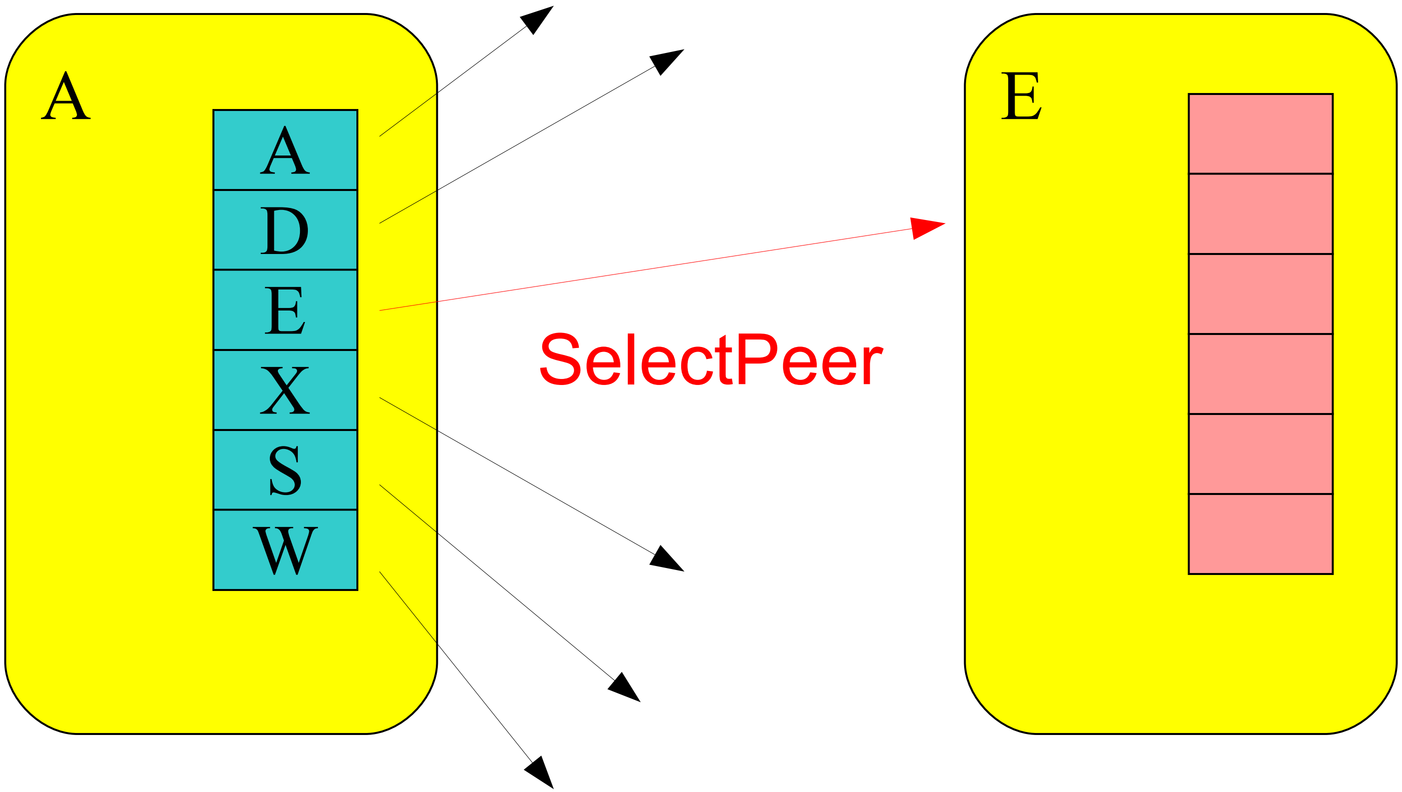
An overlay network



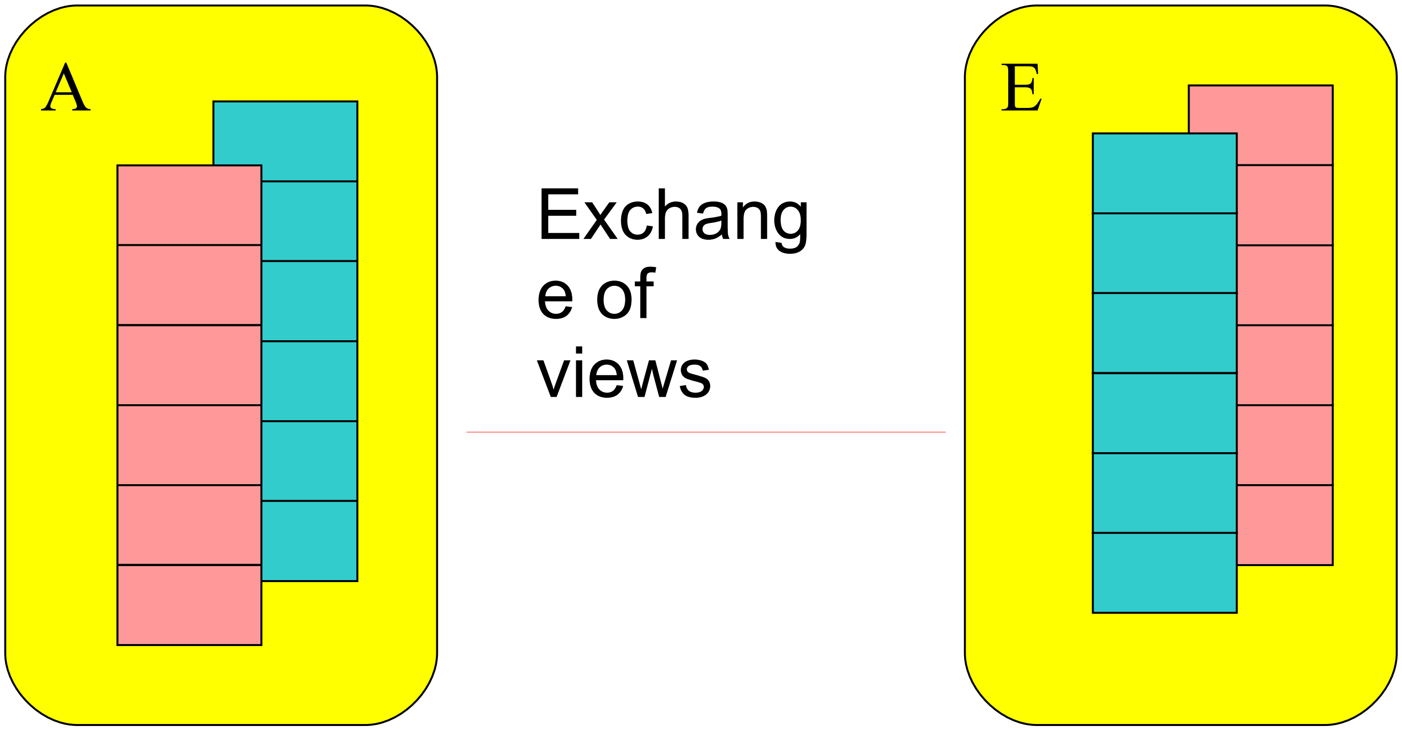
Gossip protocols for topology management



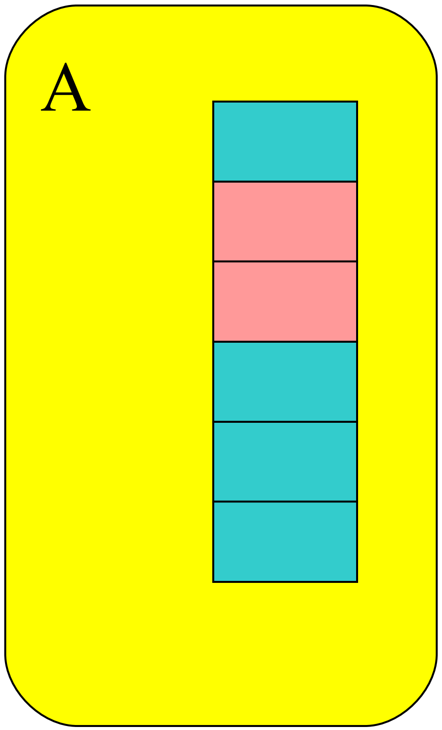
Gossip protocols for topology management



Gossip protocols for topology management

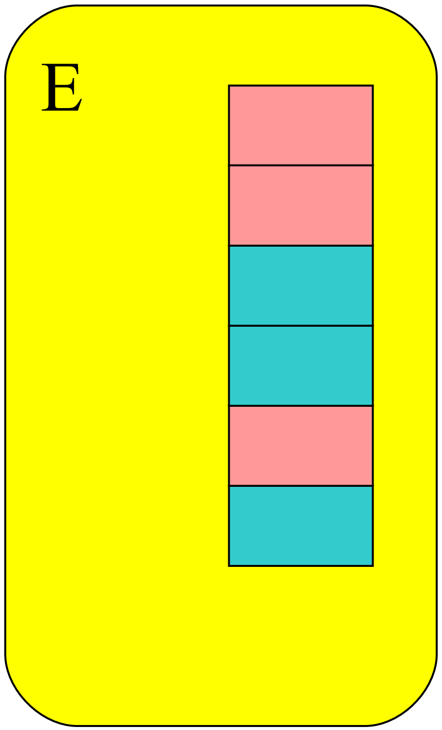


Gossip protocols for topology management



Both sides apply update

thereby redefining topology



Gossip based peer sampling

- in reality a huge number of variations exist
  - timestamps on the overlay links can be taken into account: we can select peers with newer links, or in update we can prefer links that are newer
- these variations represent important differences w.r.t. fault tolerance and the quality of samples
  - the links at all nodes define a random-like overlay that can have different properties (degree distribution, clustering, diameter, etc)
  - turns out actually not really random, but still good for gossip

Gossip based topology management

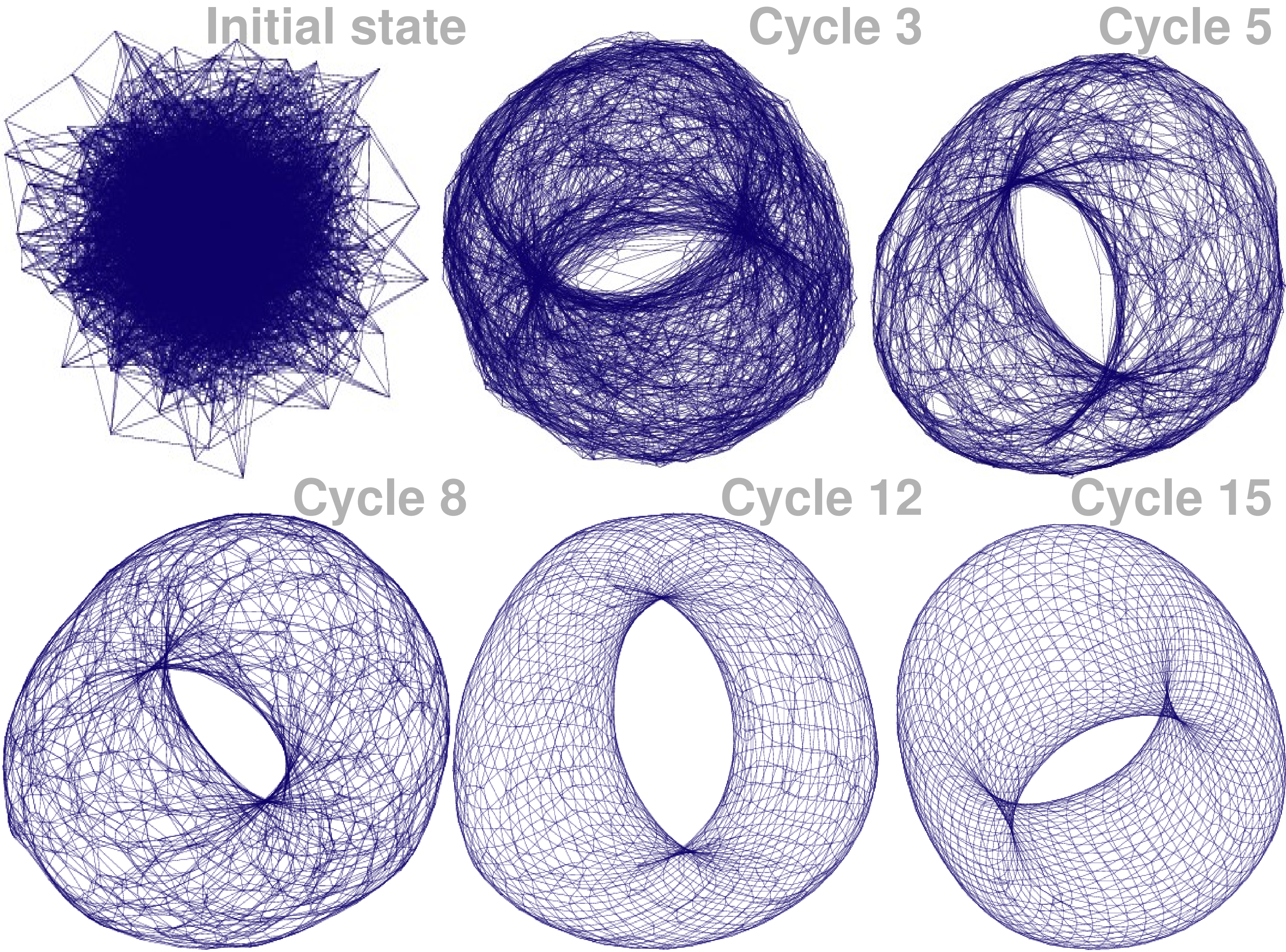
- We saw we can build random networks. Can we build any network with gossip?
- Yes, many examples
  - proximity networks
  - DHT-s (Bamboo DHT: maintains Pastry structure with gossip inspired protocols)
  - semantic proximity networks
  - etc

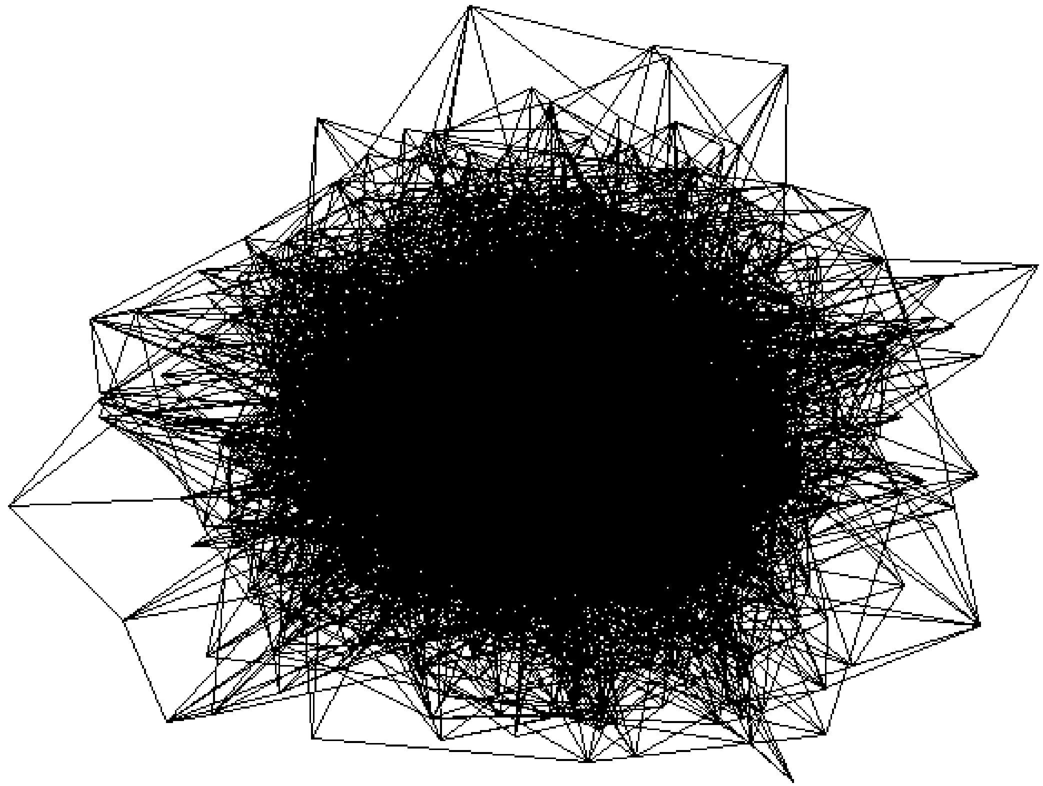
T-Man

- T-MAN is a protocol that captures many of these in a common framework, with the help of the ranking method:
  - ranking is able to order any set of nodes according to their desirability to be a neighbor of some given node
  - for example, based on hop count in a target structure (ring, tree, etc)
  - or based on more complicated criteria not expressible by any distance measure

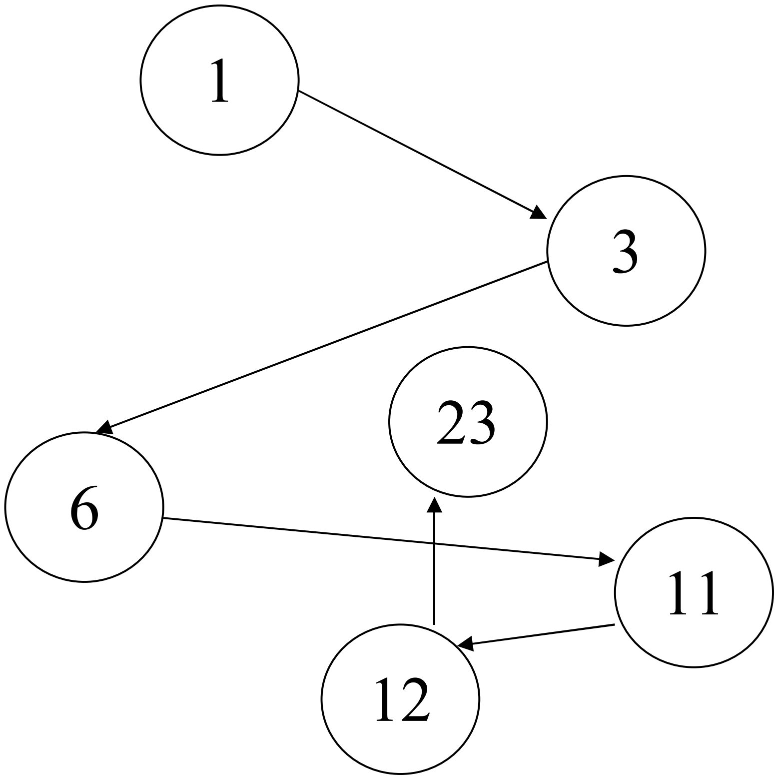
Gossip based topology management

- basic idea: random peer samples are provided by a gossip algorithm: the peer sampling service
- state: a set of overlay links to peers
- selectPeer: select the peer from the known set of peers that ranks highest according to the ranking method
- update: keep those links that point to nodes that rank highest

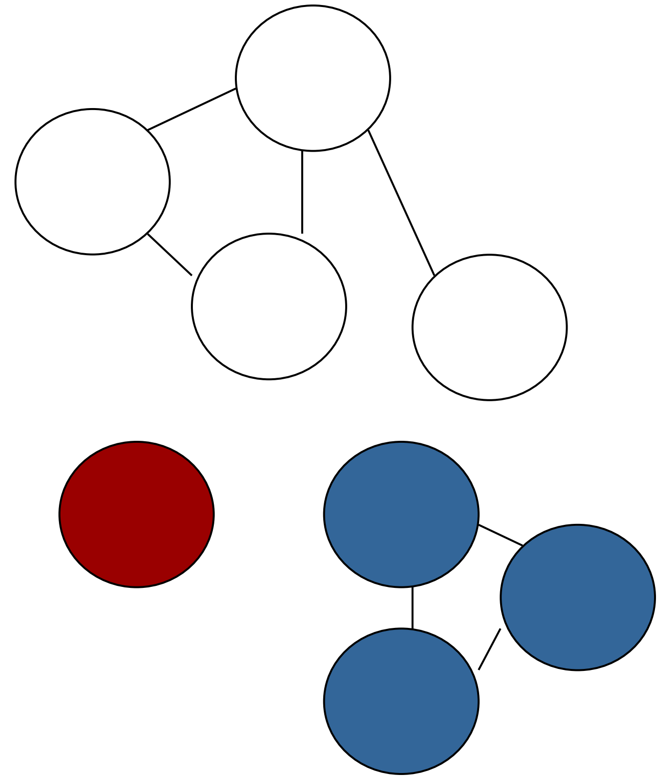

Initial state
Cycle 3
Cycle 5
Cycle 8
Cycle 12
Cycle 15



What can be bootstrapped?

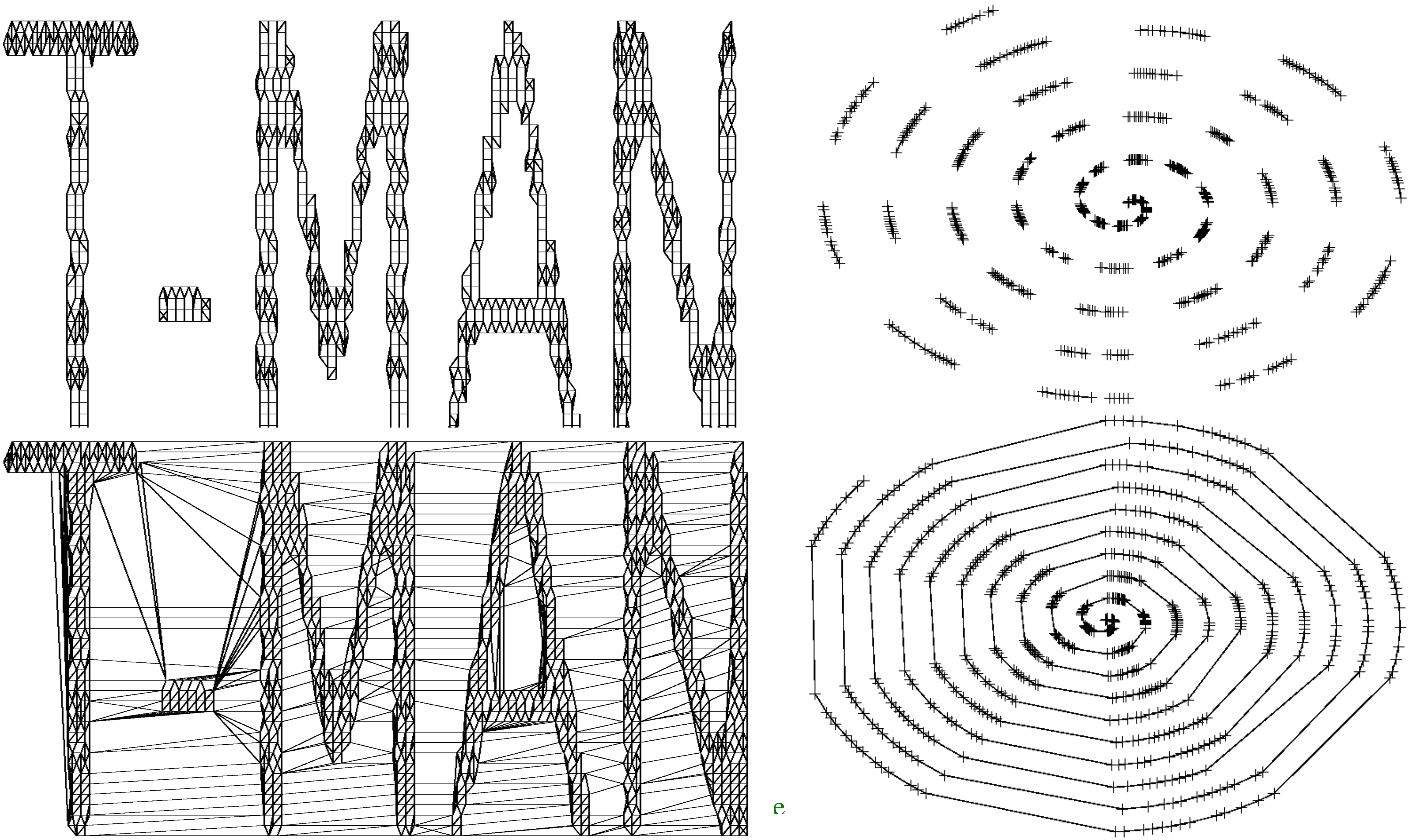


Sorting



Clustering

Illustration of clustering and sorting



Aggregation

- Calculate a global function over distributed data
  - eg average, but more complex examples include variance, network size, model fitting, etc
- usual structured/unstructured approaches exist
  - structured: create an overlay (eg a tree) and use that to calculate the function hierarchically
  - unstructured: design a stochastic iteration algorithm that converges to what you want (gossip)
- we look at gossip here

Implementation of aggregation

- state: current approximation of the average
  - initially the local value held by the node
- selectPeer: a random peer (based on peer sampling service)
- updateState(s_1,s_2)
  - $(s_1+s_2)/2$: result in averaging
  - $(s_1 s_2)^{1/2}$: results in geometric mean
  - $\max(s_1,s_2)$: results in maximum, etc

Illustration of averaging

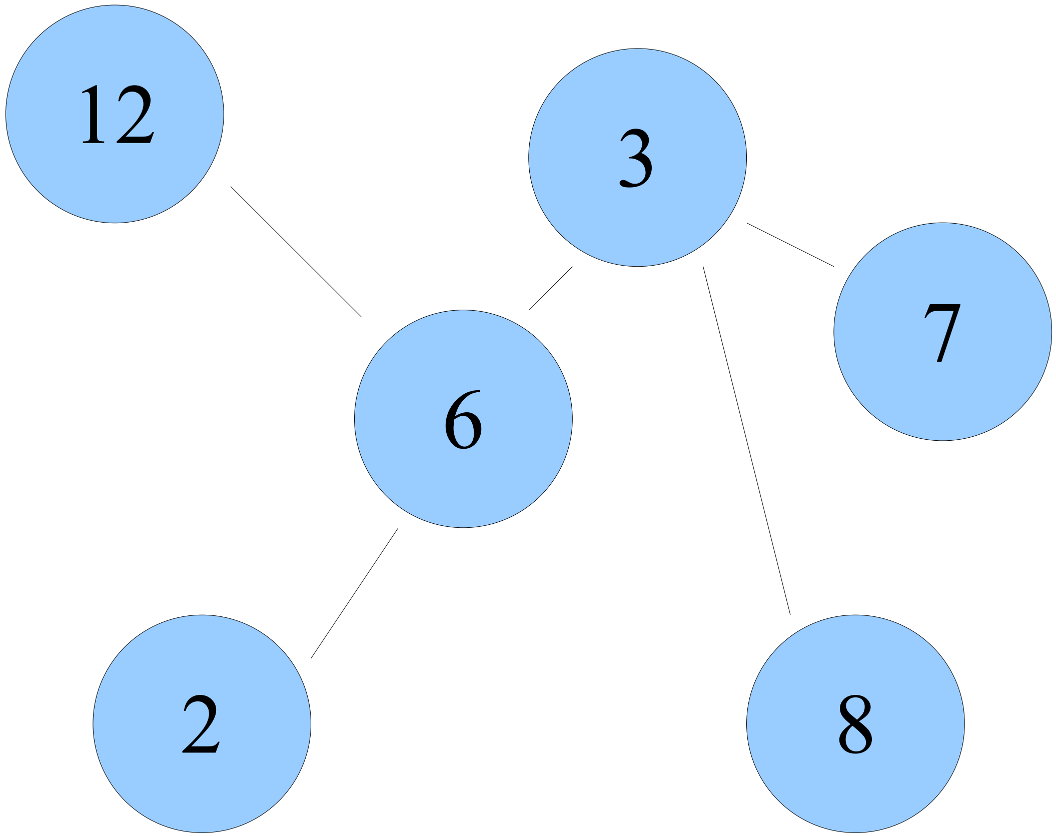
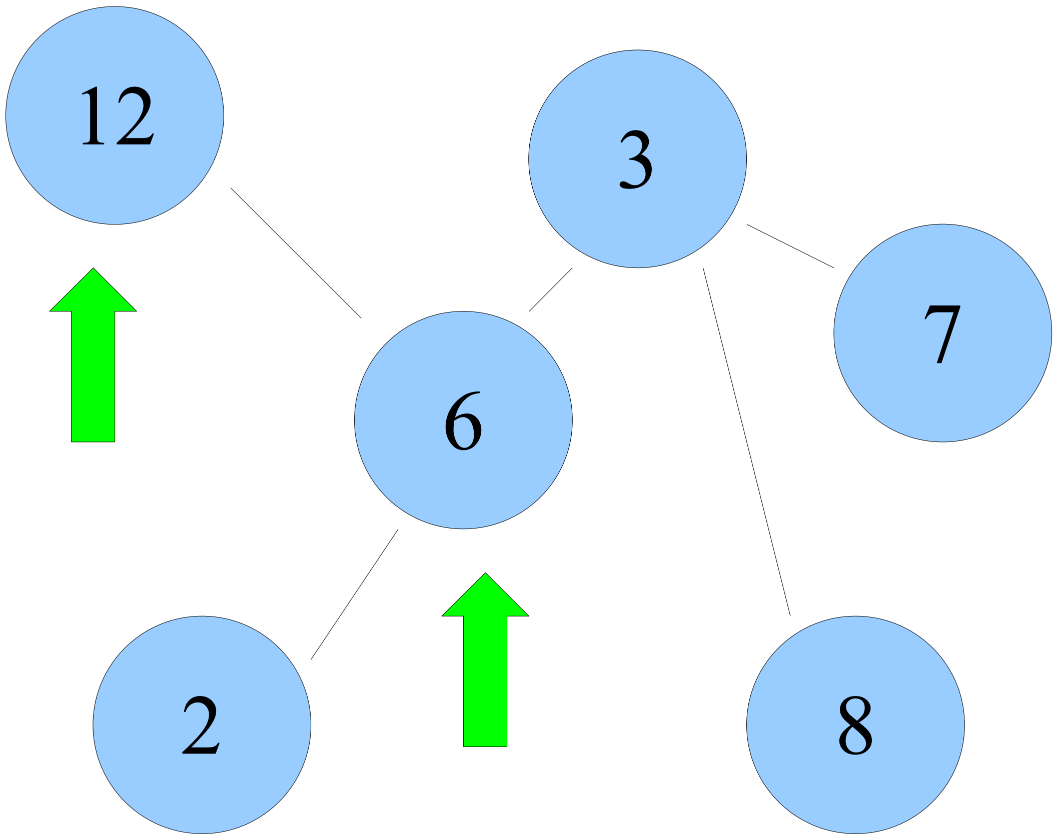


Illustration of averaging



(12+6)/2=9

Illustration of averaging

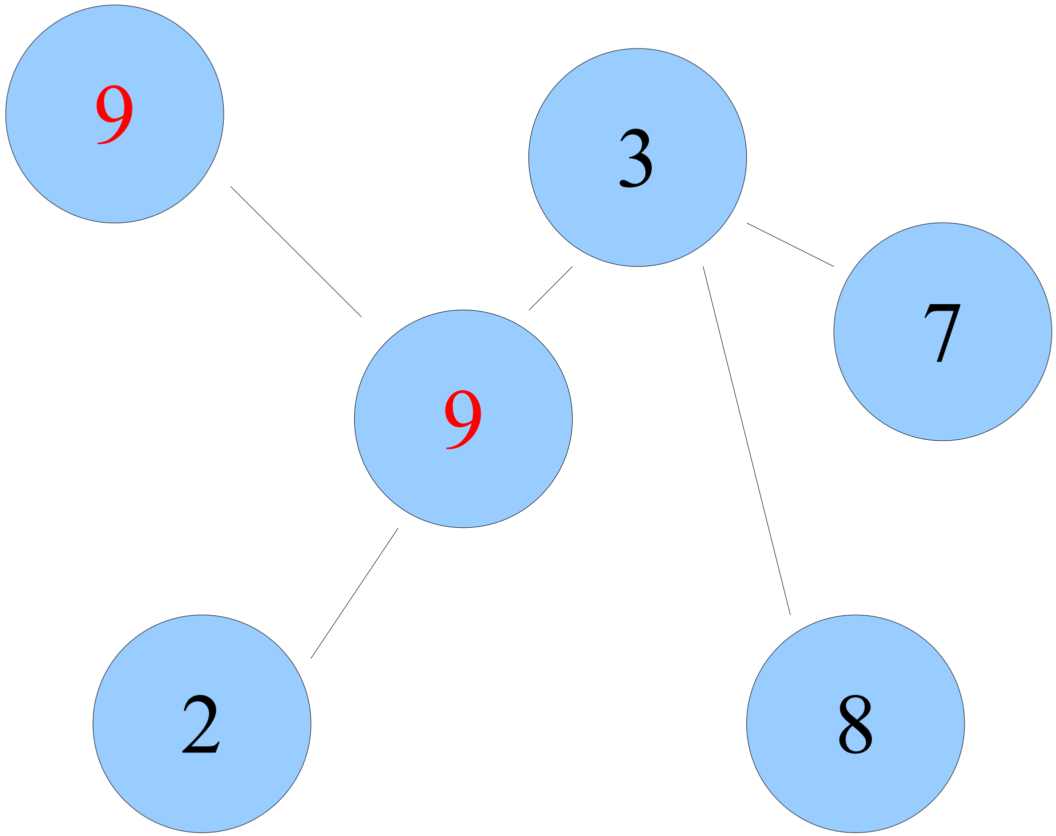
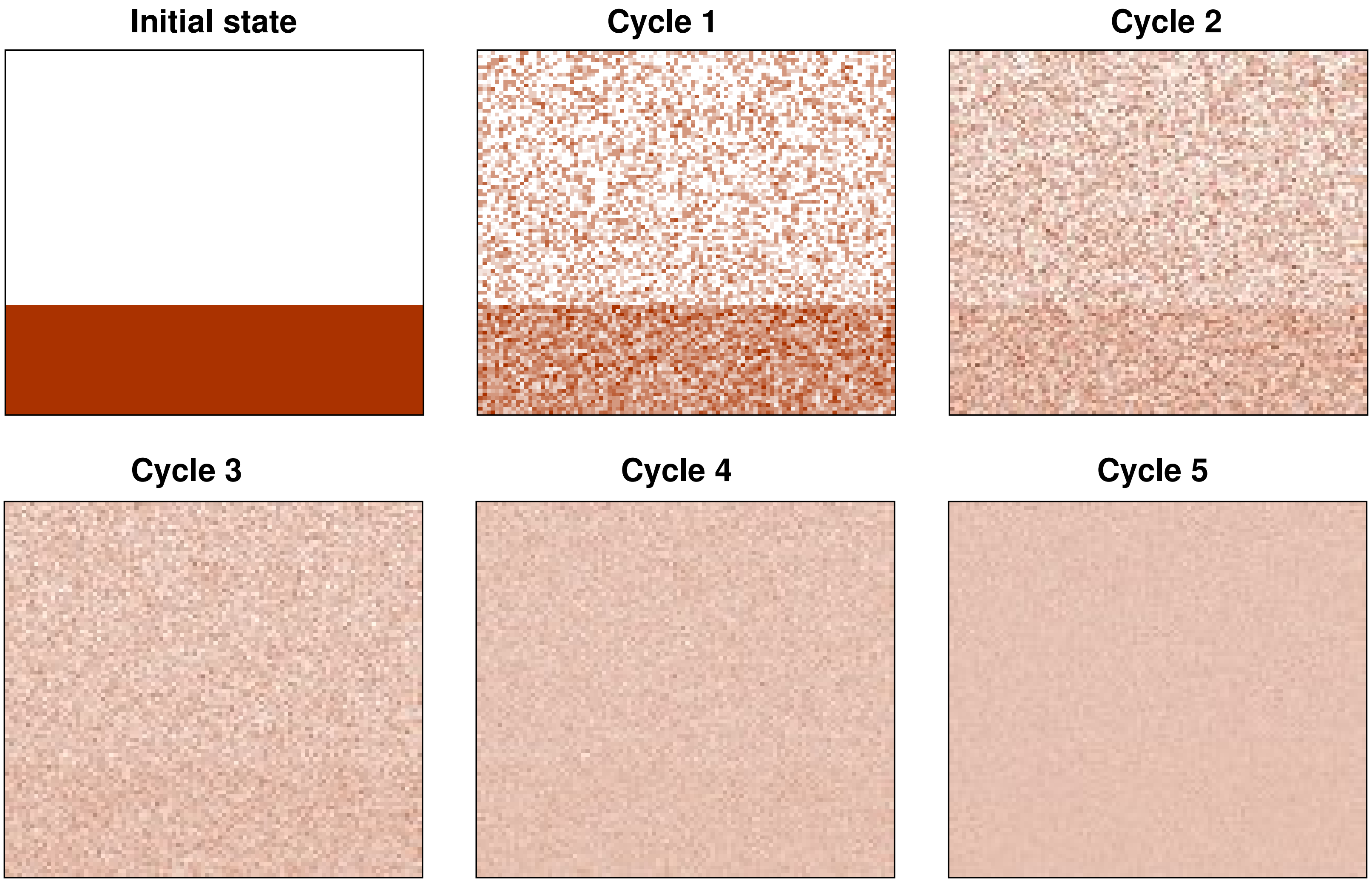
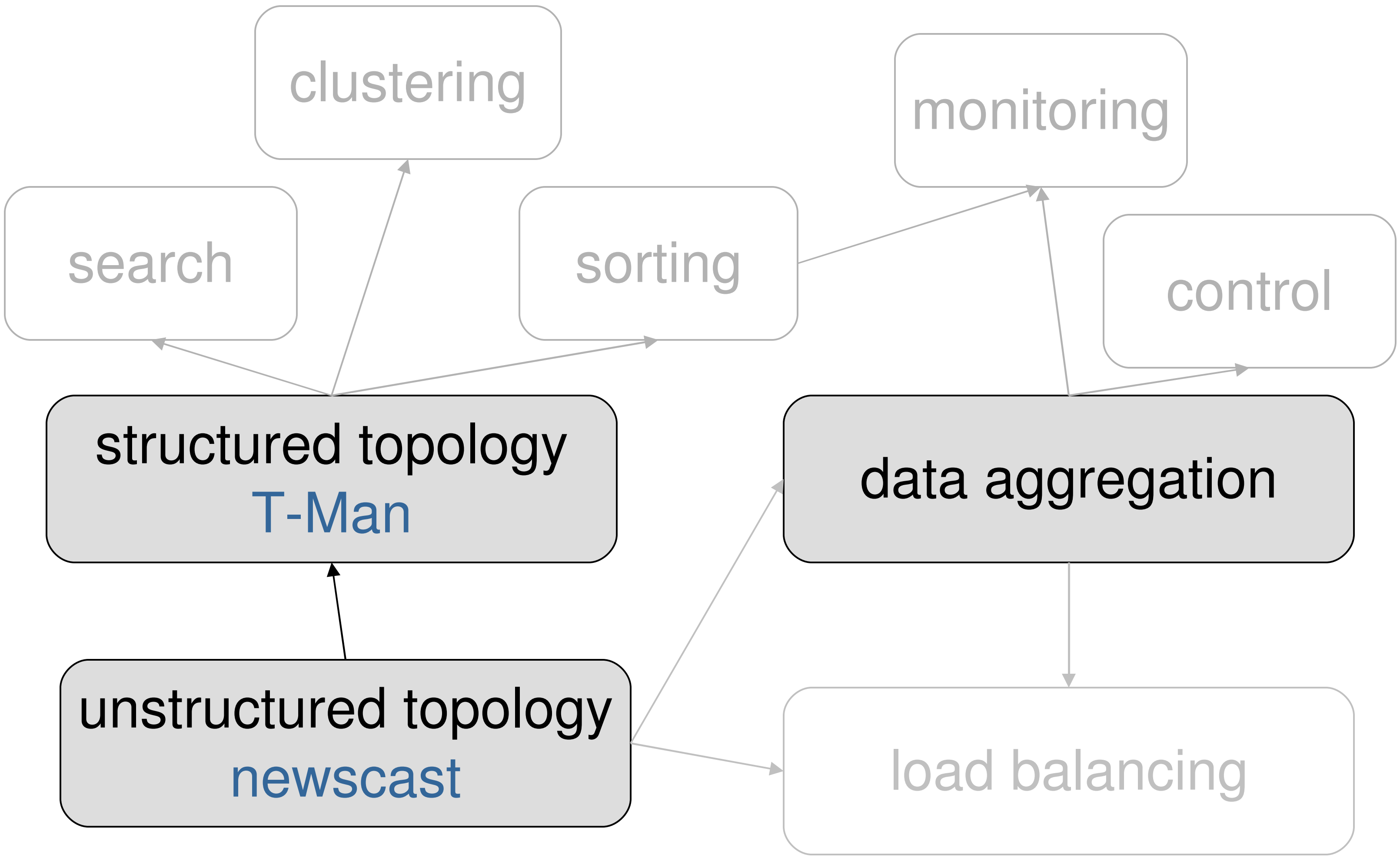


Illustration of averaging



Dependency of gossip based components



An example application

- Parallel heuristic optimization (eg genetic algorithms) on large grid
- Many possibilities for our approach eg
  - base selection on global statistics (calculated using aggregation)
  - implement a competition mechanism for alternative algorithms to automatically find the best algorithm for a given problem

Competition of algorithms

$$s := \sum_{i=1}^{n_s} \frac{1}{M_i}, \qquad P_i := \alpha \frac{1}{sM_i} + (1-\alpha)\frac{1}{n_s}$$

- M_i is a performance measure of alg. i; alpha is to avoid an unbalanced distr.
- all nodes calculate this distribution locally using aggregation (on top of peer sampling) and switch algorithms locally
- other applications where competition needs to be implemented

Toward a generic infrastructure

- P2P
  - ad hoc group
    - **independent**
    - **uncontrolled**
  - fixed purpose
  - very large-scale
  - dynamic
  - unreliable

- Grid
  - more control over resources
  - general purpose (middleware)
  - not as large-scale
  - not as dynamic
  - not as unreliable

Crossover

- P2P
  - ad hoc group
    - **independent**
    - **uncontrolled**
  - fixed purpose
  - very large-scale
  - dynamic
  - unreliable

- Grid
  - more control over resources
  - general purpose (middleware)
  - not as large-scale
  - not as dynamic
  - not as unreliable

Scenario from a user's point of view

- User prepares application against an API
    - **API contains calls to services such as search, multicast, aggregation, monitoring, etc**
- User is assigned a P2P network which supports the services needed by the application
    - **assigned network is a real P2P network: dynamic, unreliable, maybe very large-scale, etc**
    - **but with the required services and contracts**
- User executes the application
- The P2P network that was assigned is recycled

How to support this?

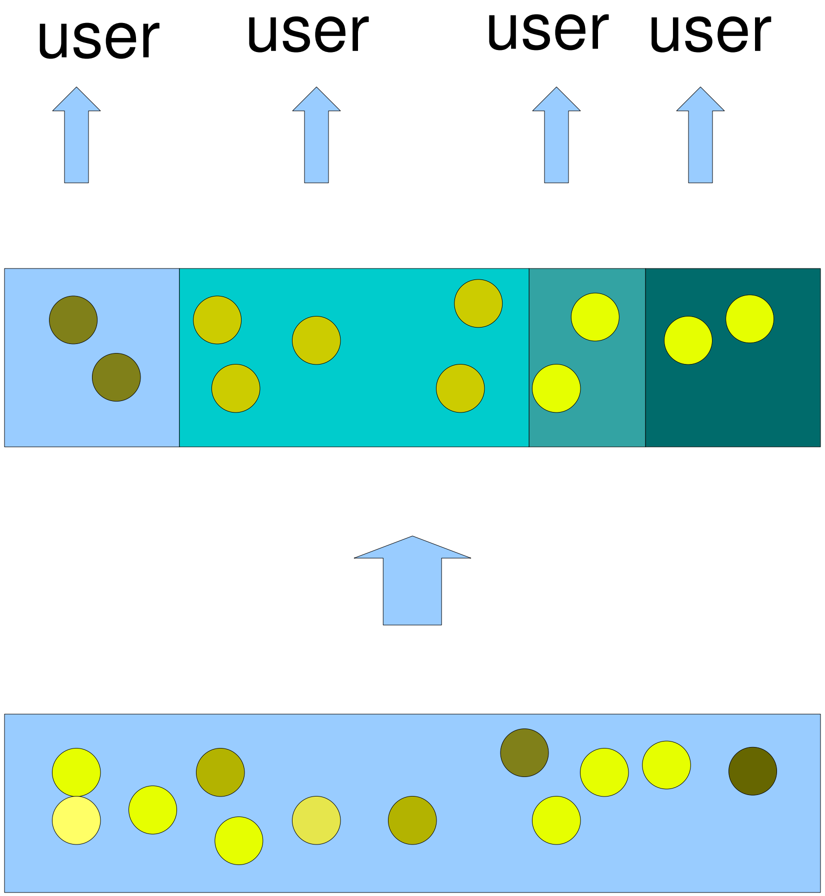
- A global pool needs to be maintained
- Partitions of this pool need to be separated and assigned
- The infrastructure that implements the services over the given partition needs to be built
- The application of the user needs to be executed
- Cleanup needs to be performed

The global pool

- requirements
  - provide a group abstraction
  - minimal functionality but
  - extremely reliable and robust
- solution
  - peer sampling service abstraction

The creation of partitions

- partitions also implement peer sampling service
  - recursive structure
- partitions are abstract and maintained (filled in) dynamically by actual set of nodes

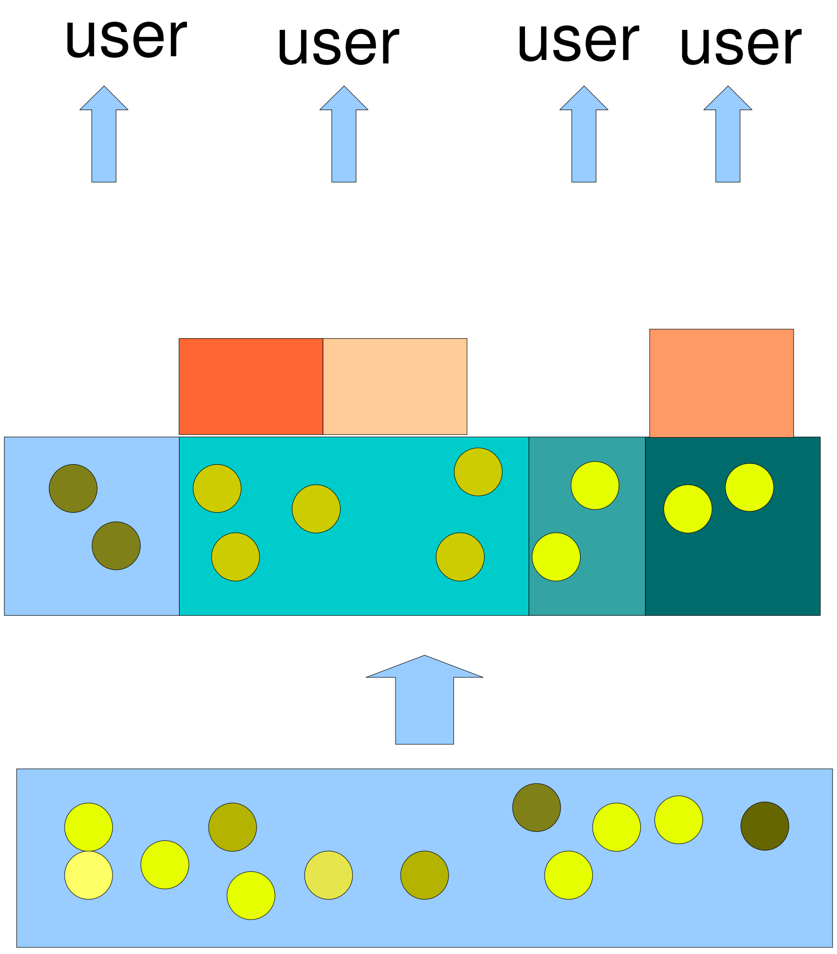


The creation of partitions

- in a self-organizing way using gossip-based slicing (not discussed here)
- Partitions can be
  - constantly created and dissolved on demand
  - even when the set of partitions is constant, they can change due to
    - **churn**
    - **changing properties of nodes**
  - The main challenge is this dynamism

Building the architecture

- the applications will need services, such as a DHT or proximity overlay
- we need to build those on top of the respective partitions (ie, the peer sampling service)



Building the architecture

- achieved through the bootstraping service
  - P2P services are based on overlay networks
  - the bootstrapping service must build arbitrary overlay networks (semantic, proximity, several DHT-s, etc) quickly, reliably and cheaply
  - should rely only on the peer sampling service
- we implement it based on a gossiping scheme as well, very similarly to the peer sampling service (T-Man)

Executing the application

- just vague ideas
  - gossip-based content distribution to spread the clients
  - start them all in a reasonable interval (self-organizing synchronization)
  - monitoring, administration, etc...

Cleaning up

- The easiest: do nothing
  - all constructs are disposable, the nodes involved in the partition will automatically join other partitions when their partition is removed

Summary

- Massively large scale systems need innovative techniques to cope with scale and other requirements
  - Eg Amazon already uses gossip algorithms, etc
- We have outlined some ideas on how to build relatively complex, but still self-organizing (and therefore robust, cheap) applications out of components
- We suggested some ideas for a generic infrastucture as well