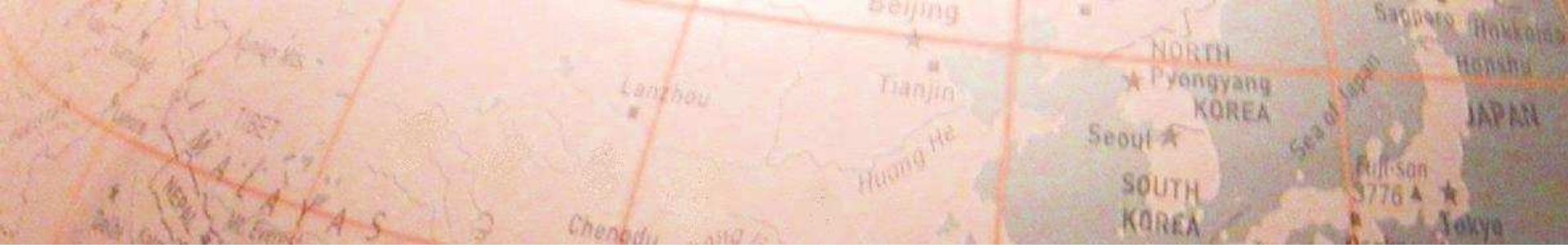


Churn Tolerance Improvement Techniques in an Algorithm-neutral DHT

Kazuyuki Shudo

Tokyo Institute of Technology
(Tokyo Tech)

Overlay Weaver

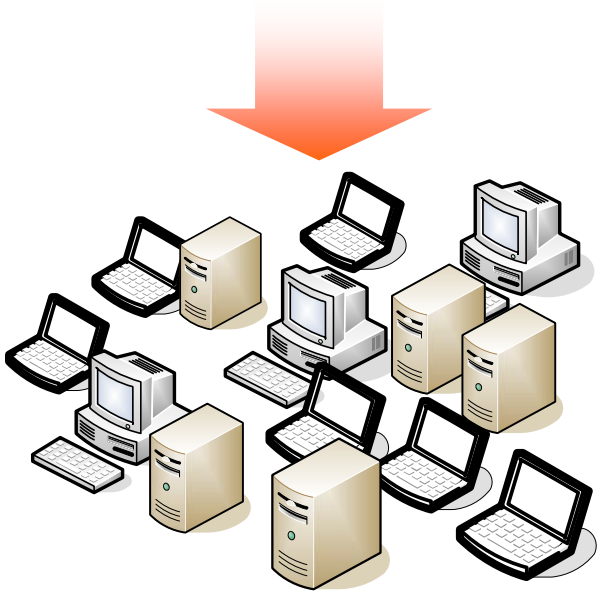
Distributed Hash Table (DHT)

- Pure P2P key-value store
  - Equal nodes construct it in decentralized and autonomous ways with no centralized node.
- Broad kinds of applications
  - Name resolutions, …
  - cf. dbm, Berkeley DB, …
  - E.g. DNS
    - “A Comprehensive Study of the DNS Design with DHT-Based Alternatives”, Proc. INFOCOM 2006.
  - E.g. Amazon’s Dynamo
    - “Dynamo: Amazon’s Highly Available Key-value Store”, Proc. SOSP 2007.
    - It’s a no-hop DHT.

put (key, value)

get (key)

remove (key, …)



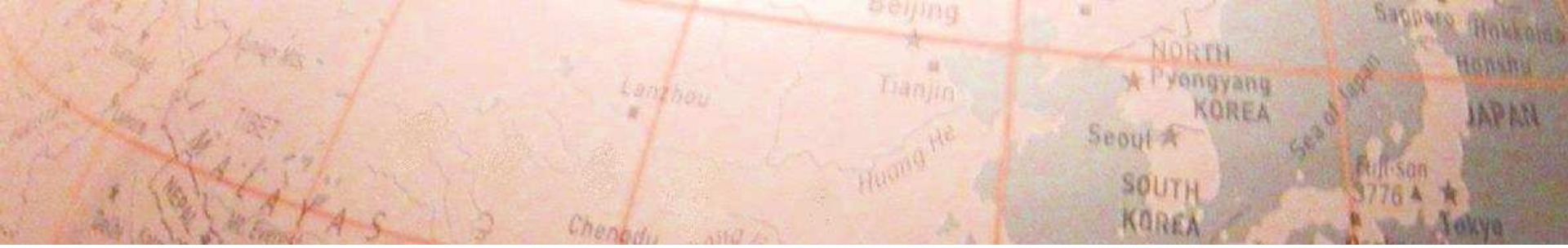
Churn

- The continuous process of node arrival and departure (join and leave)

- Decentralized distributed (peer-to-peer) systems should be tolerant to churn.
  - (Desirable) advantages of peer-to-peer systems
    - Low management and providing costs
    - High scalability
    - Moderate or high reliability
  - Approach
    - XXX Reliable nodes -> higher costs
    - OOO Accepts lower reliability and availability.
      Supposes churn. Nodes leaves and join an overlay.

Summary

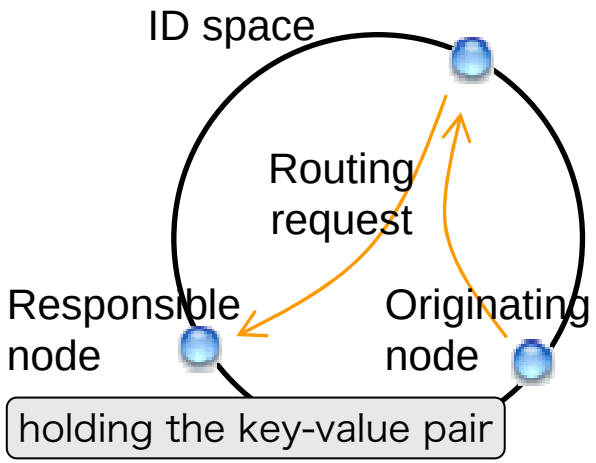
- Present churn-tolerance improvement techniques for DHTs and their effects.
- Features of those techniques
  - They do not depend on routing algorithms under the DHT layer.
    -> They work with any algorithm.
    - Chord, Kademlia, Pastry, Tapestry, Koorde, …
      Each has its merits and demerits.
- Contributions
  - Gives an empirical proof of algorithm-neutral churn-tolerance improvement techniques.
    - I have implemented them in Overlay Weaver and measured the number of successful requests.



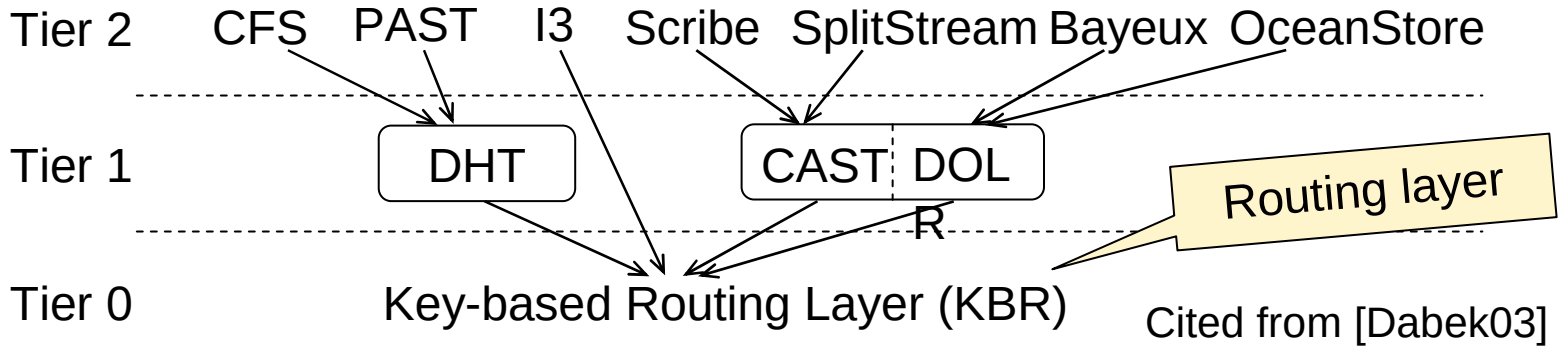
Churn problems

DHT: an application of structured overlays

- Along an abstraction in which a DHT is an application of structured overlays
  - **Put**: a node routes/forwards a request to the responsible node of the given key and passes a key-value pairs to the node.
  - **Get**: a node routes/forwards a requests to the responsible node of the given key and get a value from the node.



An abstraction of structured overlays by Dabek et al.

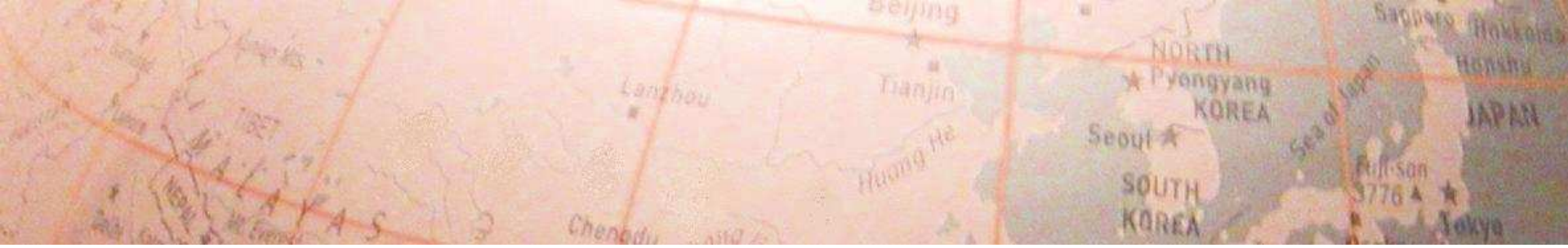


Cited from [Dabek03]

Churn problems

- A node cannot get a key-value pair which has been put -> get failed

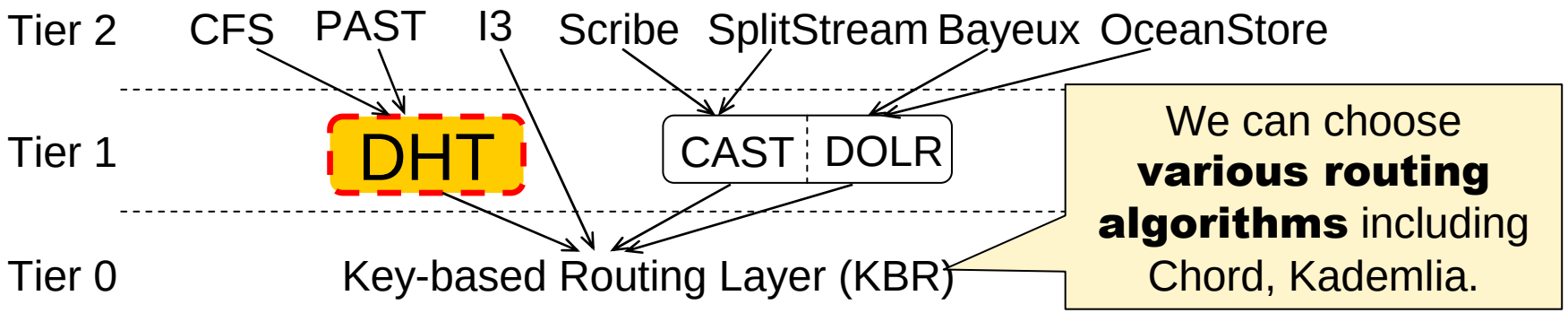
- Causes
  - The key-value pair disappeared after the put (by node departures).
  - The responsible node does not have the key-value pair.
    - Another node joined and became a new responsible node.
    - Routing to put did not reach the responsible node (due to incomplete routing tables and others).
  - A request-relaying node left the overlay network. (Note that this happens in recursive routing.)



Churn-tolerance improvement techniques

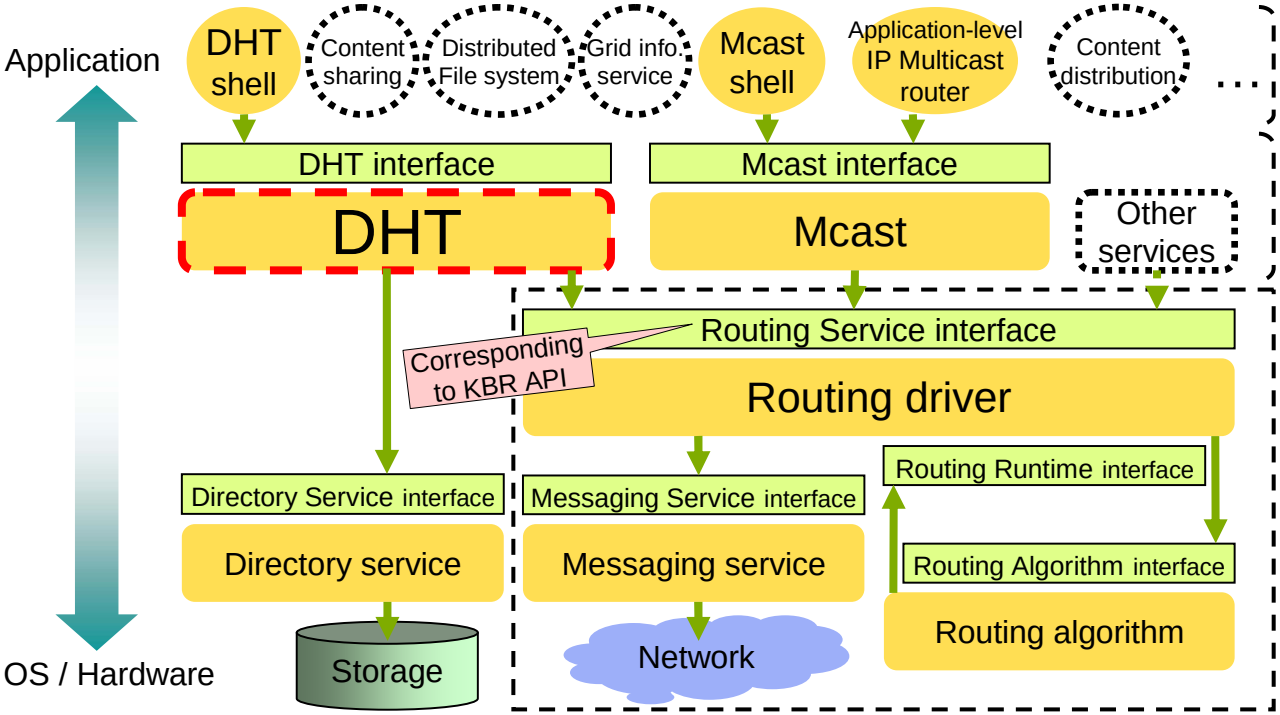
Techniques and effect measurement

- Implemented 4 techniques in the DHT layer.
  - Replication
  - Join-time transfer
  - Multiple get
  - Repeated implicit put
- Features of those techniques: They work with various routing algorithms because they are implemented in the DHT layer.
- Measured effects of those techniques with various routing algorithms.
  - Contributions: gives an empirical proof of algorithm-neutral churn-tolerance improvement techniques.

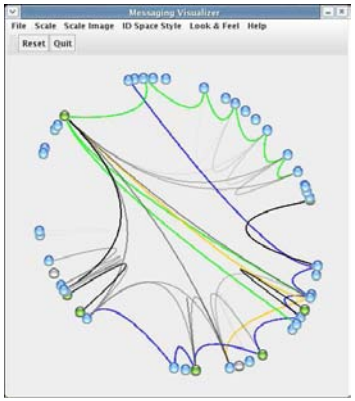


Overlay Weaver implementation

- Overlay Weaver [Shudo08]
  - A library of / a research platform for structured overlays.
  - Adopts an abstraction by Dabek et al.
  - Supports multiple routing algorithms and facilitates algorithm implementation.
    - It includes Chord, Kademlia, Pastry, Tapestry, Koorde implementations.
  - It works on a real network and emulates over 300,000 node on a single computer.
  - It runs on PlanetLab with about 580 nodes and emulates DNS.



Visualization tool

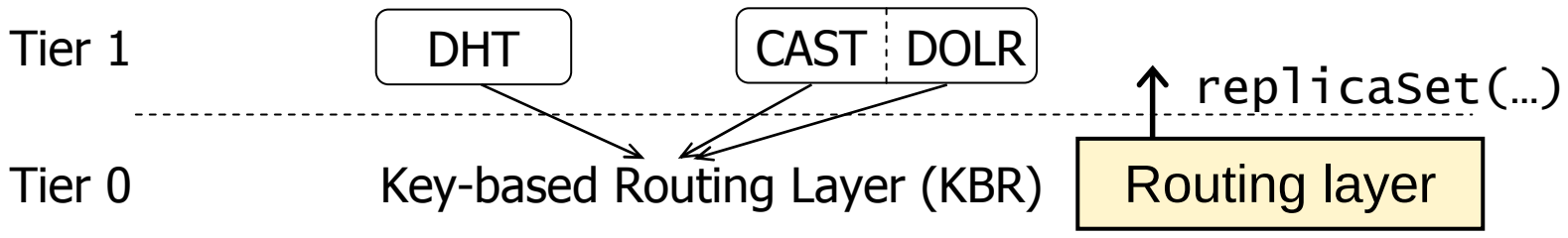


Preparation

- Routing layer returns a **list** of candidates for the responsible node as a routing result.
  - The order reflects how are the nodes adequate to be the responsible nodes.
  - 3 techniques (out of 4) utilize this list.

- In Dabek's abstraction, the routing layer provides `replicaSet(key, max_rank)`. But
  - It is only for replication.
  - With no (empirical) proof.
  - Locally called and completes on a single node.

Dabek's abstraction



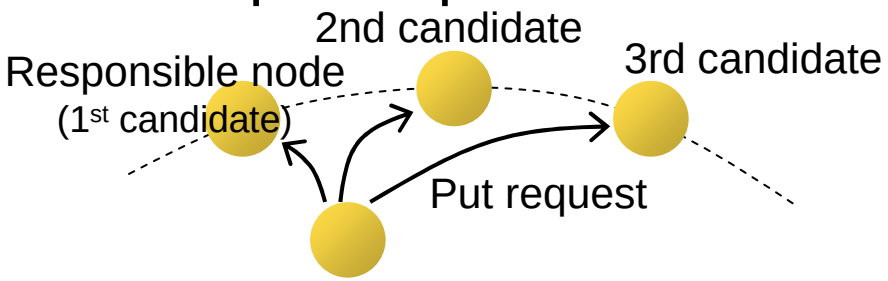
Implemented techniques

- Replication
  - Replicates a key-value pair on multiple nodes when it is put.
- Join-time transfer
  - A joining node receives key-value pairs from other nodes.
- Multiple get
  - A node requests a key to multiple nodes, not only the responsible node.
- Repeated implicit put
  - A node puts key-value pairs it holds to the DHT autonomously.

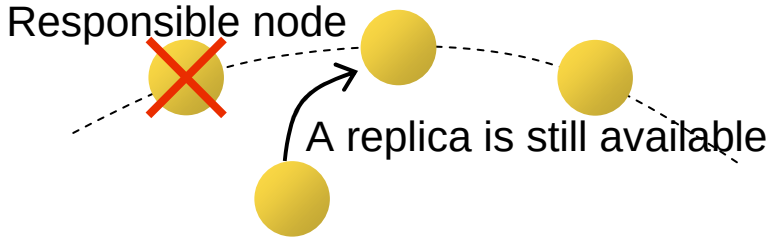
Replication

- Places a key-value pair on candidates for the responsible node when it is put.
  - Effects
    - Part of replicas are available after the responsible node has left.
  - Parameters
    - The number of replicas.
    - Which makes replicas, the originating node or the responsible node?

When a pair is put



After the responsible node has left

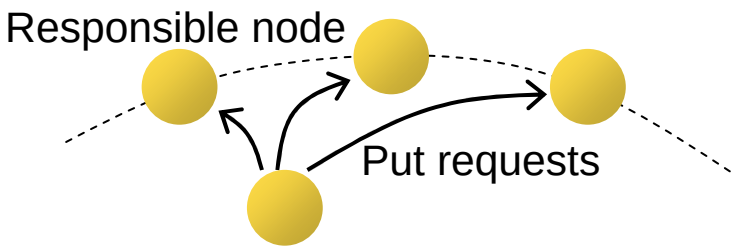


Considerations on replication

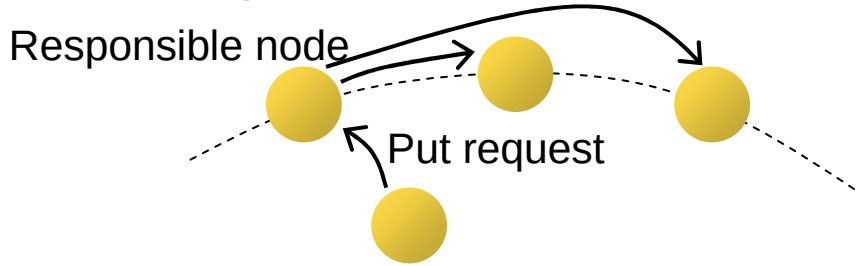
- It is possible to derive IDs for replicas from the put key.
  - E.g. (1) key, (2) key XOR 010.. (3) key XOR 100.. (4) key XOR 110..
  - This method increases the number of routing and traffic -> not adopted

- Which makes replicas, the originating node or the responsible node?
  - The number of times of communication is identical for both methods.
  - Efficiency differs in case node's ID and network proximity are related: PIS (Proximity Identifier Selection)
    -> not different because PIS is not adopted.

Originating node makes replicas

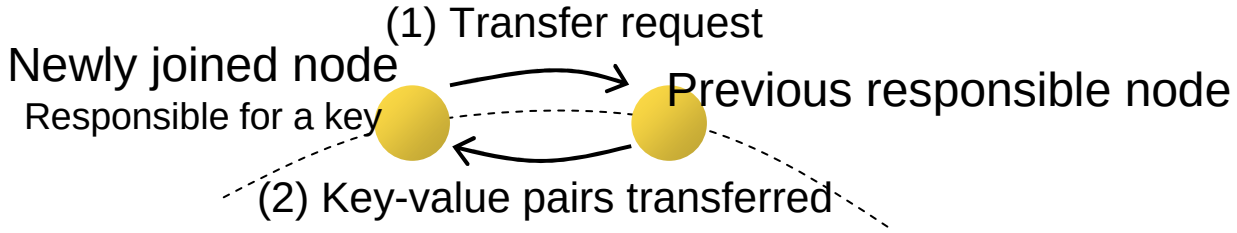


Responsible node does



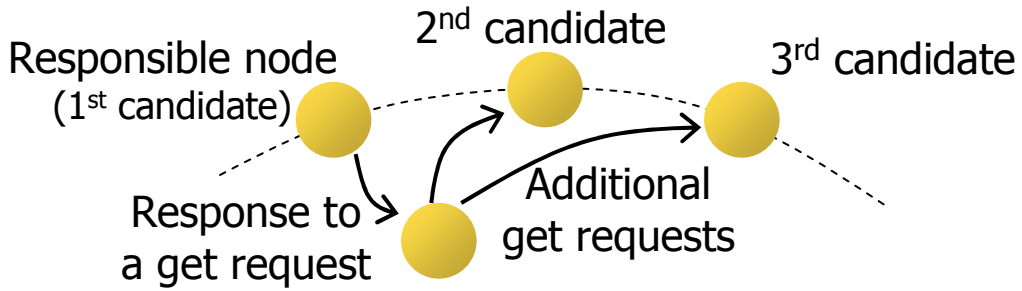
Join-time transfer

- Existing nodes transfer adequate key-value pairs to a newly joined node.
  - Method
    - A newly joined node asks candidates for the node responsible for ID of the joined node to transfer. Asked nodes transfer key-value pairs which the joined node is responsible for to the joined node.
  - Effects
    - A new responsible node which joined after the key-value pair was put can return the pair.
  - Parameters
    - The number of nodes to be asked to transfer.



Multiple get

- A node originating a get request ask multiple nodes.
  - Effects
    - It is possible to get a key-value pair which a node newly joined after the pair was put is responsible for.
      - The originating node can ask the old responsible node.
      - Join-time transfer has the same effect.
    - This technique can compensate for incomplete routing for a put request.
      - It is possible for routing not to reach the responsible node due to incomplete routing table.
  - Parameters
    - The number of nodes asked.



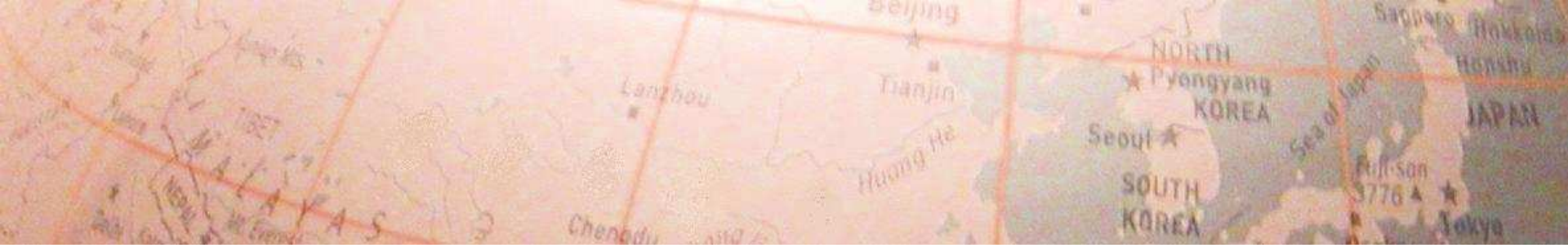
Repeated implicit put

- Each node occasionally puts key-value pairs it has onto the DHT.
  - Makes replicas as same as the usual puts.
  - This is the only technique which does not utilize the list of candidates for the responsible node directly.
  - Effects
    - Fills replicas, which decreases as nodes leave the overlay.
    - The same effect as join-time transfer. The new responsible node can have pairs. But the effect is limited.
      - Implicit puts have an interval -> transfer by implicit put involves a time lag.
      - This technique has the same effect as join-time transfer even if the number of replicas is 1, though replicas cannot be filled.
  - Parameters
    - Time interval
      - Randomly fluctuated not to be synchronized between nodes.

Causes of get failure each technique deals with

- **Replication**: prevents replicas from disappearing.
- **Repeated implicit put**: fills up replicas.
- **Join-time transfer** and **multiple get**: let the newly joined responsible node have key-value pairs.

Techniques / Causes	Replication	Join-time transfer	Multiple get	Repeated implicit put
Disappeared key-value pairs	√			√ (requires replication)
Responsible node does not hold the pair				
Newly joined responsible node		√	√	√ (effects limited)
Nodes on a route left			√	√



Effects

How much each technique improve the rate of successful get requests ?

Experiments

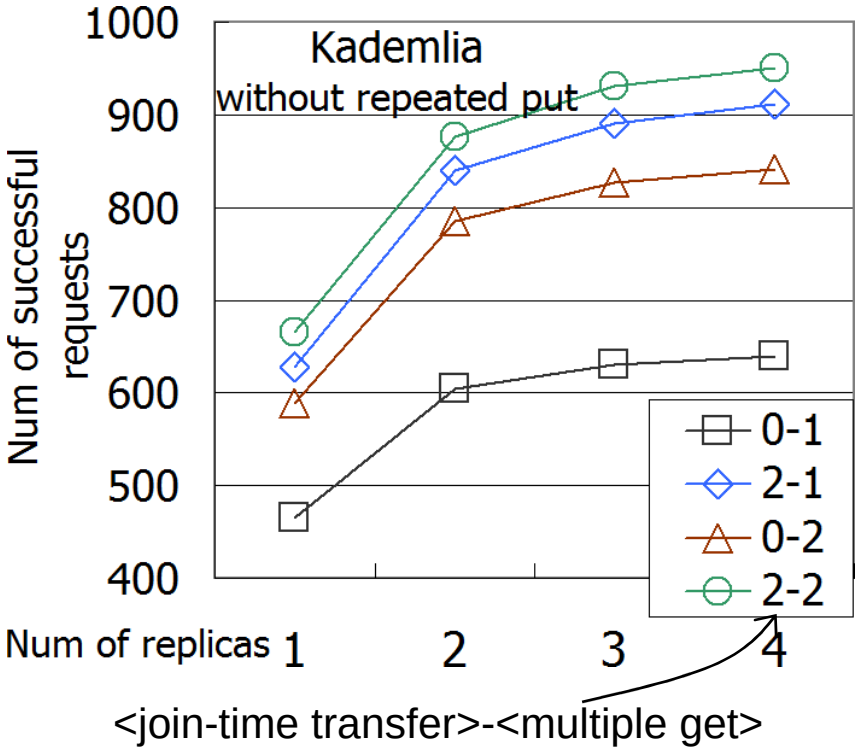
to measure effects of each technique

- On a single PC,
  - 1,000 (virtual) nodes run
  - with churn
  - The number of successful get requests were counted.
- Targets
  - 5 routing algorithms Overlay Weaver support.
  - Iterative and recursive routing.
  - All combinations of them.
- Experiment scenario
  - invokes 1,000 nodes
  - lets all nodes join a DHT, one every 0.15 sec
  - puts different 1,000 key-value pairs on the DHT, one every 0.2 sec
  - gets all pairs from the DHT, one every 0.2 sec
- Environment
  - Distributed environment emulator of Overlay Weaver 0.6.4
  - Java SE 5.0 Update 12 for x86
  - Linux 2.6.21 for x86-64
  - 2.8 GHz Pentium D
- Conditions
  - Random nodes did puts and gets
  - Churn lasted during puts and gets
  - Churn model: the number of nodes is constant <- a new node joins just after a node left
  - Churn frequency is 2 times / sec -> the average of node life time is 500 sec
  - Churn is timed by a Poisson process
  - Communication timeout is 3 sec and routing timeout is 10 sec
  - The interval of repeated implicit put is 30 sec

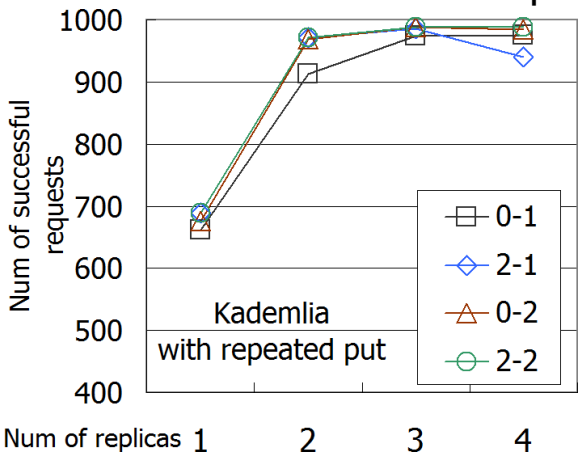
Results and observations

Every targets showed the same tendency

- Intuitive (natural) results
  - Replicas improved success rates.
  - Join-time transfer and multiple get improved success rates.
- Other observations
  - In Pastry and Kademlia, join-time transfer (# of asked nodes: 2) is more effective than multiple get (# of asked nodes: 2). Chord showed the opposite results.
  - There were cases that 3 replicas showed better results than 4 replicas.



Repeated put hided
the effects of other techniques

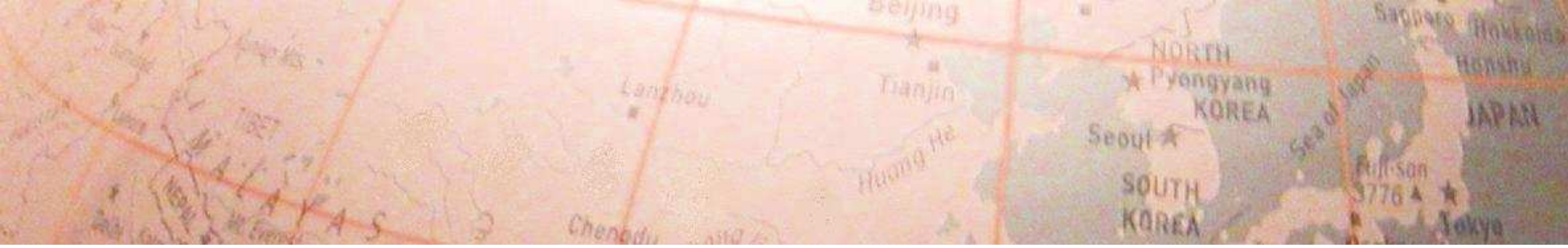


Gave an empirical proof of algorithm-neutral churn-tolerance techniques

Results and observations (cont'd)

- Part of get requests still failed due to …
  - Techniques could not cover up churn completely.
    - All replicas disappeared, …
  - A newly joined node has an incomplete routing table for some time.
    - Routing does not reach the responsible node.
  - Communication timeout happened multiple times and routing timeout happened.

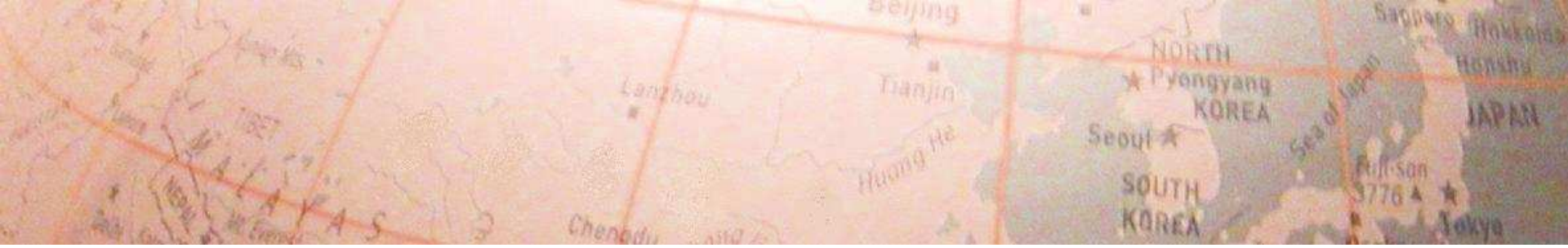
- Note: Results here do not show superiority and inferiority of routing algorithms.
  - Each algorithm has its parameters and they were fixed as the default settings of Overlay Weaver 0.6.4.
    - E.g. stabilization interval in Chord



Now we see the techniques work.

So, how do we determine

- application of each technique and
- parameters of each technique ?



Cost performance

Considerations toward future work

Cost performance

- Performance
  - presented as the rate of successful requests.
  - Function of the rate.

- Cost: what increases with the techniques
  - The number of times and traffic of communication
  - Time required to join, put and get
    - Join-time transfer and multiple get requires additional time.
  - Memory and storage consumption
    - *N* replicas consume memory / storage *N* times.
    - It can be very expensive in embedded environments.
  - Processing by CPU
    - Response, power consumption, …

Attention of existing researches

Costs heavily depend on application **environments**

System behavior depending on applications

- Each technique requires costs at its own timings.

Timing / Techniques	join	put	get	always (!)
Replication		√		
Join-time transfer	√			
Multiple get			√	
Repeated implicit put				√

  - For example,
    - Join-time transfer is cheap if join and leave are rare.
    - Repeated implicit put requires continuous costs even though neither put nor get requests.
- Each application has its behavior.
  - DNS: the number of get requests is much larger than put requests -> multiple get is expensive.

We have to consider system behavior depending on applications

Future work

- Calculate **cost** (vs. performance) taking account of applications and environments.

- Applications
  - DNS: put frequency << get frequency
  - Sensor network: put ??? Get

- Environments
  - E.g. The ratio of costs of storage and communication

Related work

- Dabek's Layered model of structured overlay [Dabek03]
  - DHT is an application of structured overlay networks
  - A function provided by the routing layer to implement replication: `replicaSet(key, max_rank)`
    - only for replication
    - without (empirical) proof.

- Churn tolerance techniques implemented in Bamboo [Rhea04]
  - Detection policy of failed nodes, timeout adjustment, proximity neighbor selection (PNS)
  - All techniques are for routing layer, not for DHT layer, independent from techniques in this research and can be combined.

- Evaluation of DHT implementations using network emulator peeremu [Kato07]
  - Authors evaluates DHT implementations including Bamboo, Chord, Accordion and FreePastry. About 1,000 nodes ran on 10 or 20 PCs. They measured the rate of successful get requests and the time required to get.

- Supernodes
  - Only nodes elected as supernodes construct an overlay (DHT). They serve other ordinary nodes.
  - This architecture relaxes churn tolerance required for a DHT.
  - Churn tolerance is still an important property.

Summary

- Presented a number of churn tolerance techniques for DHT and demonstrated their effects.
  - All techniques are neutral to underlying routing algorithms.
- Gave an empirical proof of algorithm-neutral churn-tolerance techniques.
- Considered cost-performance and its calculation.
  - To be considered
    - Applications and their environments: PC, embedded, Internet, wireless, …
    - System behavior

References

- [Dabek03] F. Dabek et al., “Towards a Common API for Structured Peer-to-Peer Overlays”, IPTPS 2003, 2003.
- [Shudo08] K. Shudo et al., “Overlay Weaver: An Overlay Construction Toolkit”, Computer Communications, 2008.
- [Rhea04] S. Rhea et al., “Handling Churn in a DHT”, USENIX ’04, 2004.
- [Kato07] D. Kato et al., “Evaluating DHT Implementations in Complex Environments by Network Emulator”, IPTPS 2007, 2007.