Peer-to-Peer Systems and Applications

Lecture 5: Selected Topics

TECHNISCHE UNIVERSITÄT DARMSTADT

Chapter 5, 8, 21 and 33:

Part II: Unstructured P2P Systems
Part III: Structured Peer-to-Peer
Part VI: Search and Retrieval Systems
Part X: Advanced Issues

*Original slides provided by Ralf Steinmetz and Vasilios Darlagiannis (Technische Universität Darmstadt), and David Hausheer and Burkhard Stiller (University of Zürich)

0. Lecture Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Selected DHT Algorithms: CAN

Design, Routing, Node Join,
Node Departure, References

*Based partially on original slides by Sylvia Ratnasamy et al. (UC Berkeley)

1.1. Design of CAN

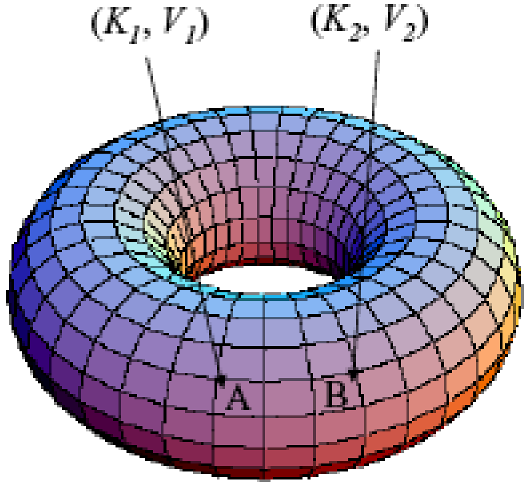
TECHNISCHE UNIVERSITÄT DARMSTADT

- CAN: A Scalable Content Addressable Network
  - Ratnasamy et al., SIGCOMM 2001

- *d*-dimensional Cartesian coordinate space (*d*-torus)

- Each node owns a *zone* on the torus

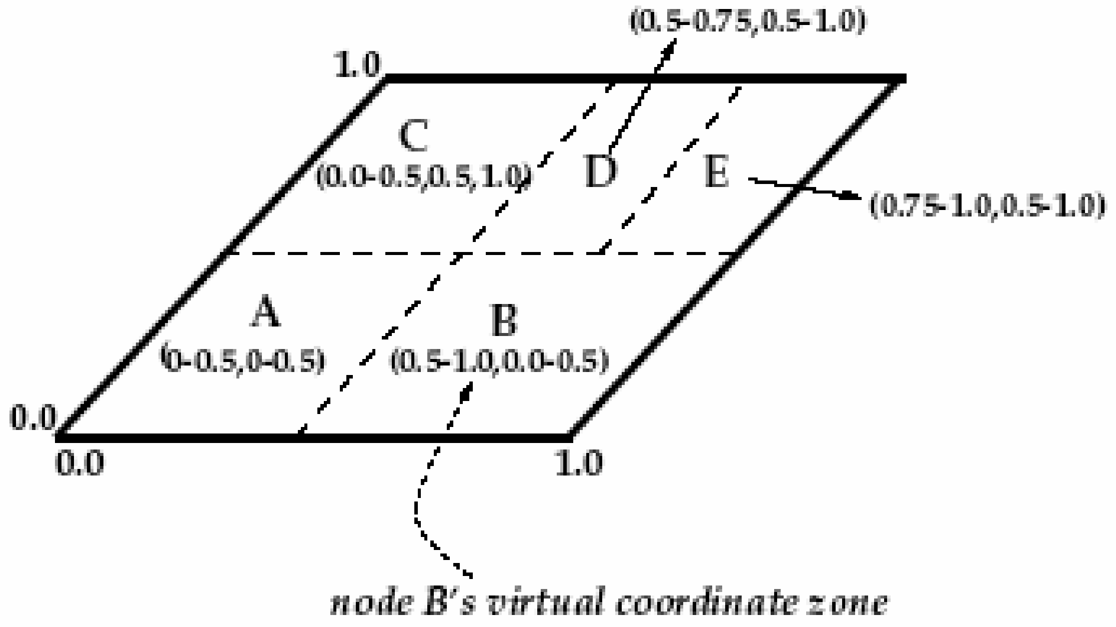
- To store key value pair (K_1, V_1),
  - K_1 mapped to point P_1 using uniform hash function
  - (K_1, V_1) stored at the node N that owns the zone containing P_1



2-torus

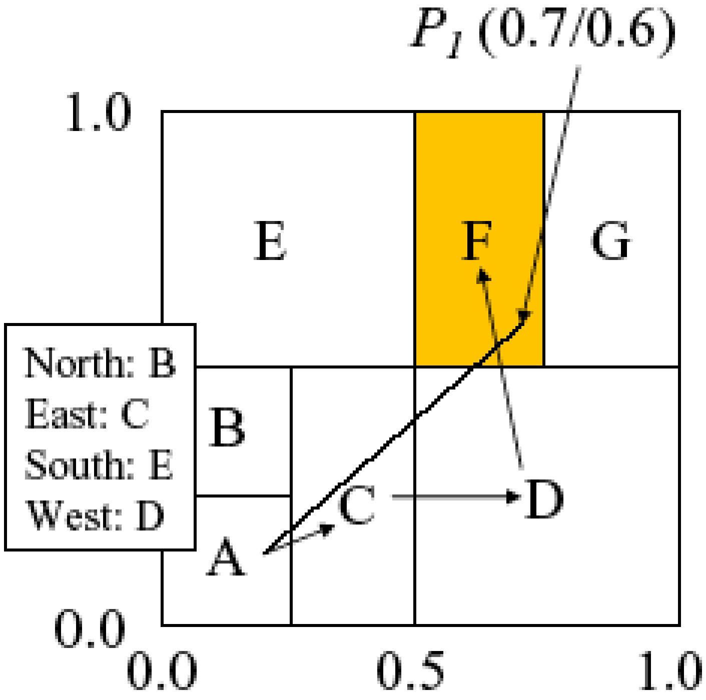
1.1. Design of CAN

- Each node stores IP address and coordinate zone of adjoining zones
- This set of neighbors is the node's routing table



1.2. CAN Routing

- How to route from node *A* to point P_1 at (0.7, 0.6)?
  - Draw straight line from point in *A*'s zone to P_1
  - Follow straight line using neighbor pointers



- For *d*-dimensional space partitioned into *n* equal zones, each node maintains 2*d* neighbors
  - Average routing path length:

$$\left(\frac{d}{4}\right)\left(n^{\frac{1}{d}}\right)$$

1.3. CAN: Node Join

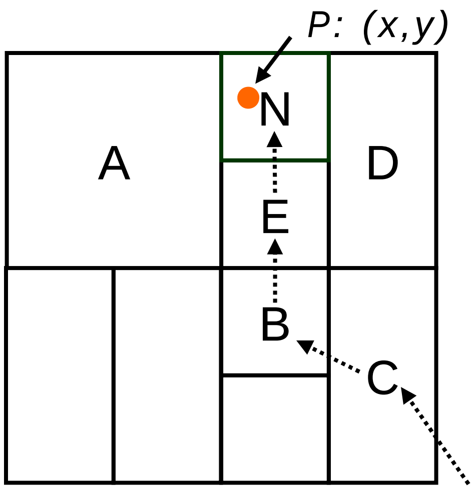
1. New node finds a node already in CAN

2. New node chooses random point P and sends JOIN message to node whose zone contains *P*, say node *N*

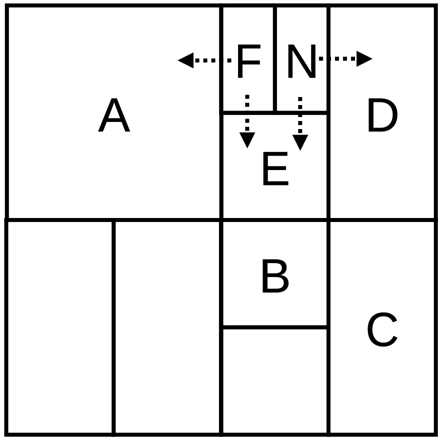
3. *N* splits its zone and allocates “half” to new node, transfer of (key,value) pairs

4. New node learns neighbor set from *N*

5. *N* updates its neighbor set to include new node



Routing path from node C to point (x,y)



1.3. CAN: Node Departure

- Graceful Node Departure
  - Node explicitly hands over zone and (key,value) pairs to one of its neighbors
  - Merge to form “valid” zone if possible
  - If not, two zones are temporarily handled by smallest neighbor

- Node Failures
  - Each node periodically sends messages to each of its neighbors
  - Nodes that detects failure initiates *takeover mechanism*
  - Takeover mechanism ensures node with smallest volume takes over the zone

1.4. References

- CAN
  - S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: A Scalable Content Addressable Network; In Proceedings of ACM SIGCOMM 2001, August 2001.
- Kademlia
  - P. Maymounkov and D. Mazieres: *Kademlia: A peer-to-peer information system based on the xor metric*; In Proceedings of IPTPS02, Cambridge, USA, March 2002.
- Viceroy
  - D. Malkhi, M. Naor, D. Ratajczak: *Viceroy: A scalable and dynamic emulation of the butterfly*; 21st ACM Symposium on Principles of Distributed Computing (PODC), 2002.
- P-Grid
  - Karl Aberer: *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*; Sixth International Conference on Cooperative Information Systems (CoopIS), 2001.

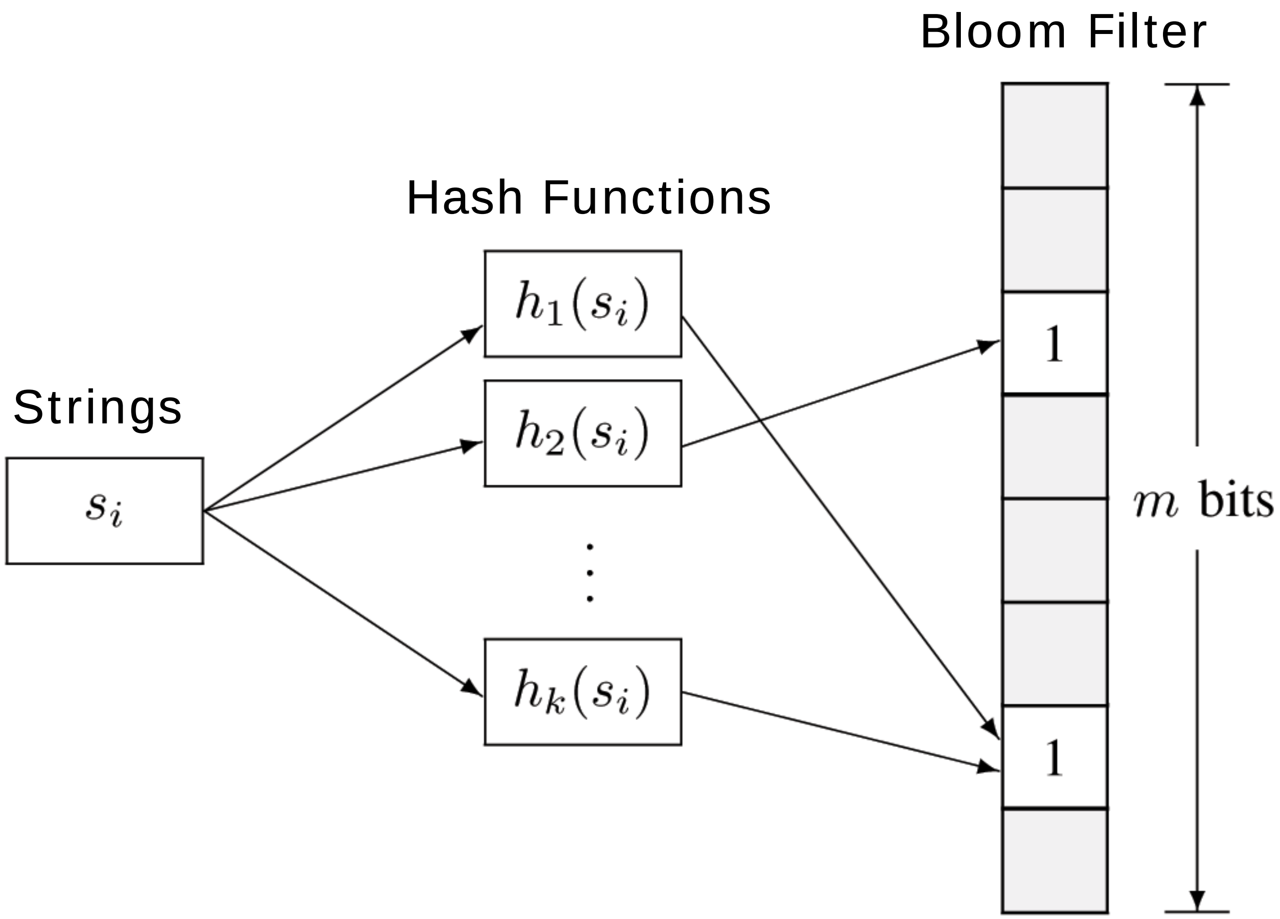
2. Bloom Filters

Traditional Bloom Filter, Attenuated Bloom Filter, References

2.1. Traditional Bloom Filter

- An array of *m* bits, initially all bits set to 0
- A bloom filter uses *k* independent hash functions
  - *h1, h2, ..., hk* with range {1, ..., m}
- Each key is hashed with every hash function
  - Set the corresponding bits in the vector
- Operations
  - Insertion
    - The bit A[hi(x)] for 1 < i < k are set to 1
  - Query
    - Yes if all of the bits A[hi(x)] are 1, no otherwise
  - Deletion
    - Removing an element from this simple Bloom filter is impossible

2.1. Insertion of an Element



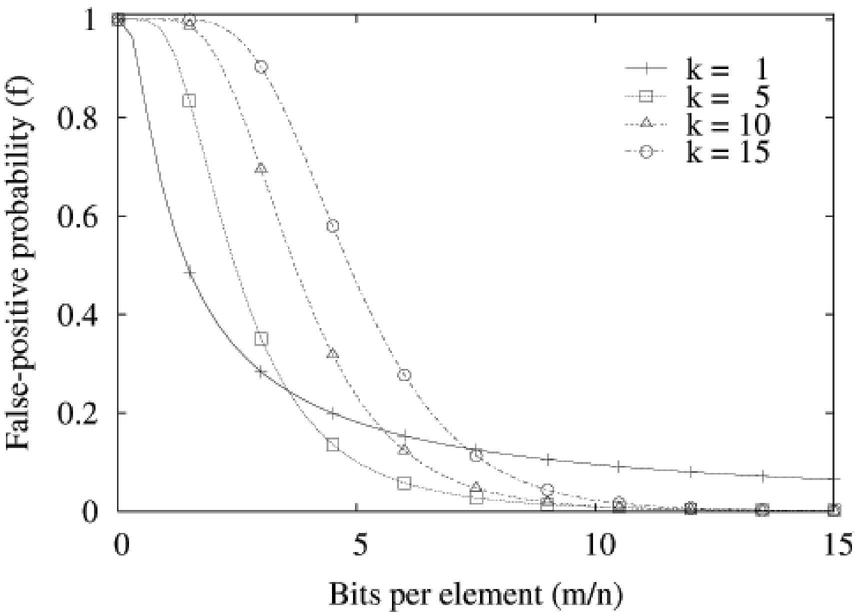
2.1. Properties

- Space Efficiency
  - Any Bloom filter can represent the entire universe of elements
    - In this case, all bits are 1
- No Space Constraints
  - Add never fails
  - But false positive rate increases steadily as elements are added
- Simple Operations
  - Union of Bloom filters: bitwise OR
  - Intersection of Bloom filters: bitwise AND

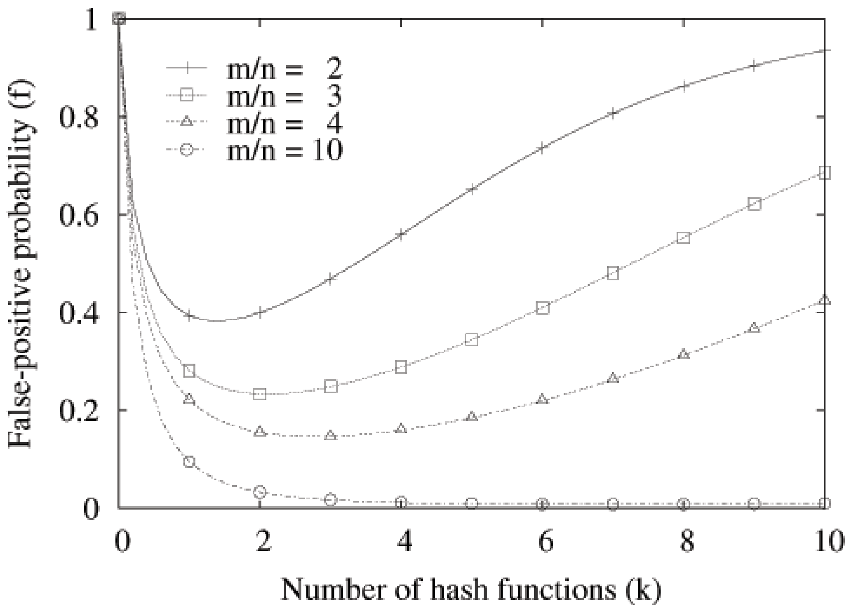
2.1. False-Positive Probability

TECHNISCHE UNIVERSITÄT DARMSTADT

- No false negative, but false positive
- False-positive probability: $f = (1 - e^{-\frac{nk}{m}})^k$
  - *n* number of strings; *k* hash functions; *m*-bit vector



=> A longer bit vector and fewer insertions are always better



=> Given m/n, there is an optimal number of hash functions (f = 0.5^k) (when 50% of the bits are set)

2.1. Examples

- Application Example
  - Hash every attribute of a service description
  - Hash the query string
  - If the corresponding bits are set to 1, the query is satisfied

- Example for False-positives
  - Insertions
    - Hash („color printer“) => (1,4,6)
    - Hash („digital camera“) => (3,4,5)
  - Query
    - Hash („heat sensor“) => (3,4,6)
    - Matches since bits 3,4,6 are all set to 1

2.1. Applications

- Distributed Caching
- Collaboration in Overlay and Peer-to-Peer Networks
- Resource Routing
- Packet Routing
- Measurement Infrastructures
  - Traffic Flow Measurement
    - Space-Code Bloom Filter
  - Network Intrusion Detection
    - Packet Scanning
    - Identify malicious content, e.g. Internet worms and viruses

2.1. Bloom Filter Variants (1)

- Attenuated Bloom Filter
  - Use arrays of Bloom filters to store shortest path distance information
- Counting Bloom Filters
  - Each entry in the filter need not be a single bit but rather a small counter
  - Delete operation possible (decrementing counter)
- Spectral Bloom Filters
  - Extend the data structure to support estimates of frequencies
- Compressed Bloom Filters
  - When the filter is intended to be passed as a message
  - False-positive rate is optimized for the *compressed* bloom filter (uncompressed bit vector m will be larger but sparser)
  - Parameters can be adjusted to the desired trade-off between size and false-positive rate

2.1. Bloom Filter Variants (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

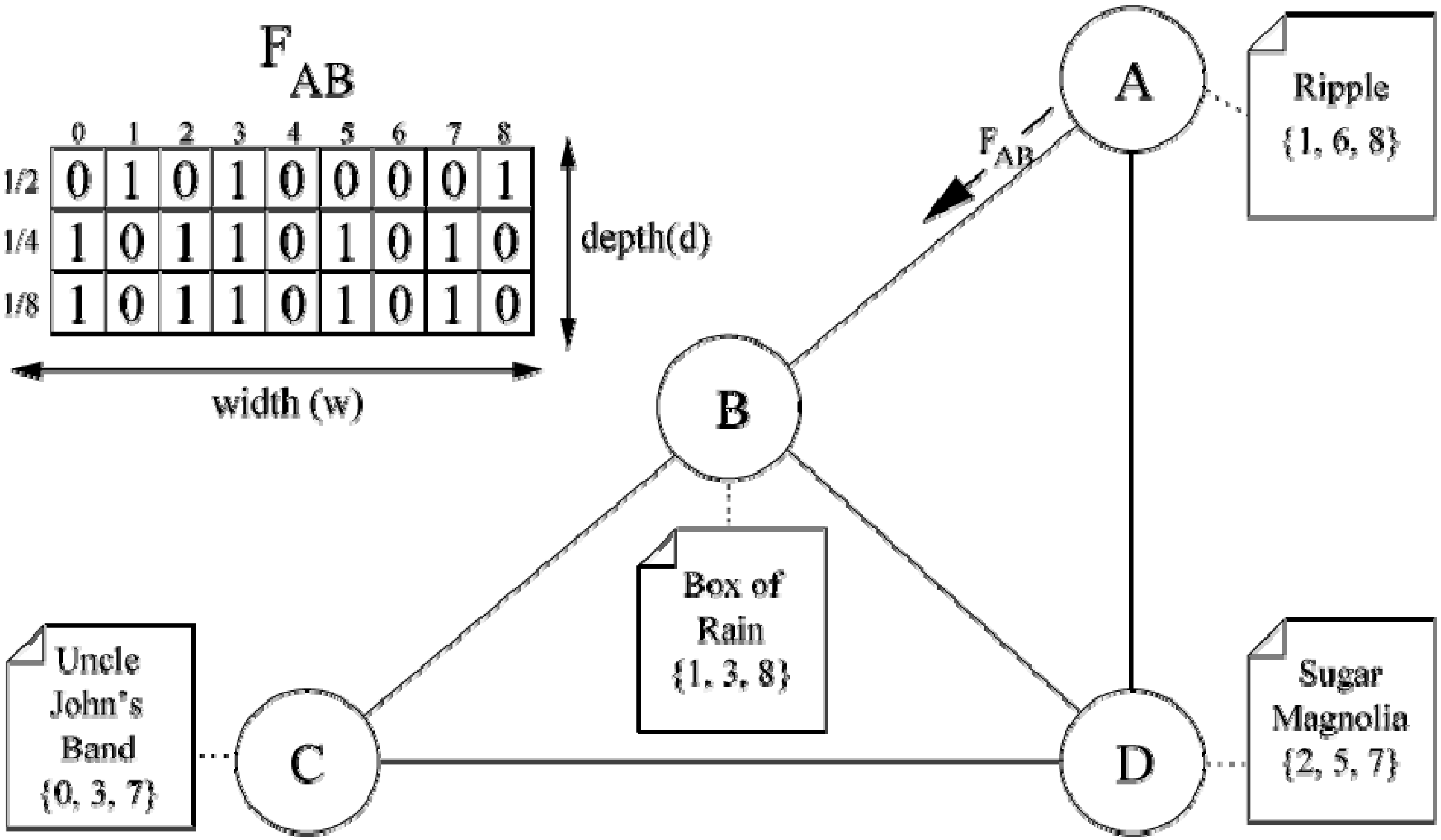
- Generalized Bloom Filter
  - Two type of hash functions *gi* (reset bits to 0) and *hj* (set bits to 1)
  - Start with an arbitrary vector (bits can be either 0 or 1)
  - In case of collisions between *gi* and *hj*, bit is reset to 0
  - Produces either false positives or false negatives
- Space-Code Bloom Filter
  - Made of l groups of hash functions, each group viewed as a traditional Bloom filter
    - **Insertion:**
      One group of hash function is selected, and the bits are set to 1
    - **Query:**
      Matches a group if all the bits of that group are set to 1; keeps track of the number of groups matched

2.2. Attenuated Bloom Filter

- Definition
  - An attenuated Bloom filter of depth *d* is an array of *d* normal Bloom filters
- Assumption
  - Each node has a set of overlay neighbors participating in the location algorithm
- Association
  - Each neighbor link is associated with an attenuated Bloom filter
- Construction
  - Bloom filter *k* is a union of objects at exactly *k* hops away

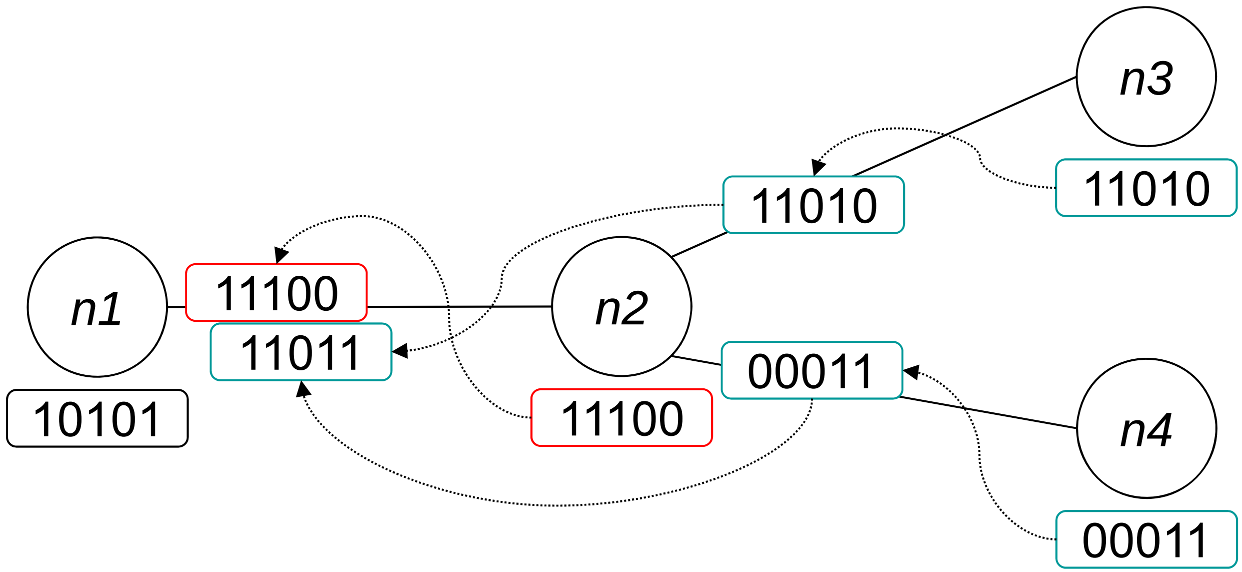
Sean C. Rhea, John Kubiatowicz: Probabilistic Location and Routing; Infocom 2002.

2.2. Example (1)

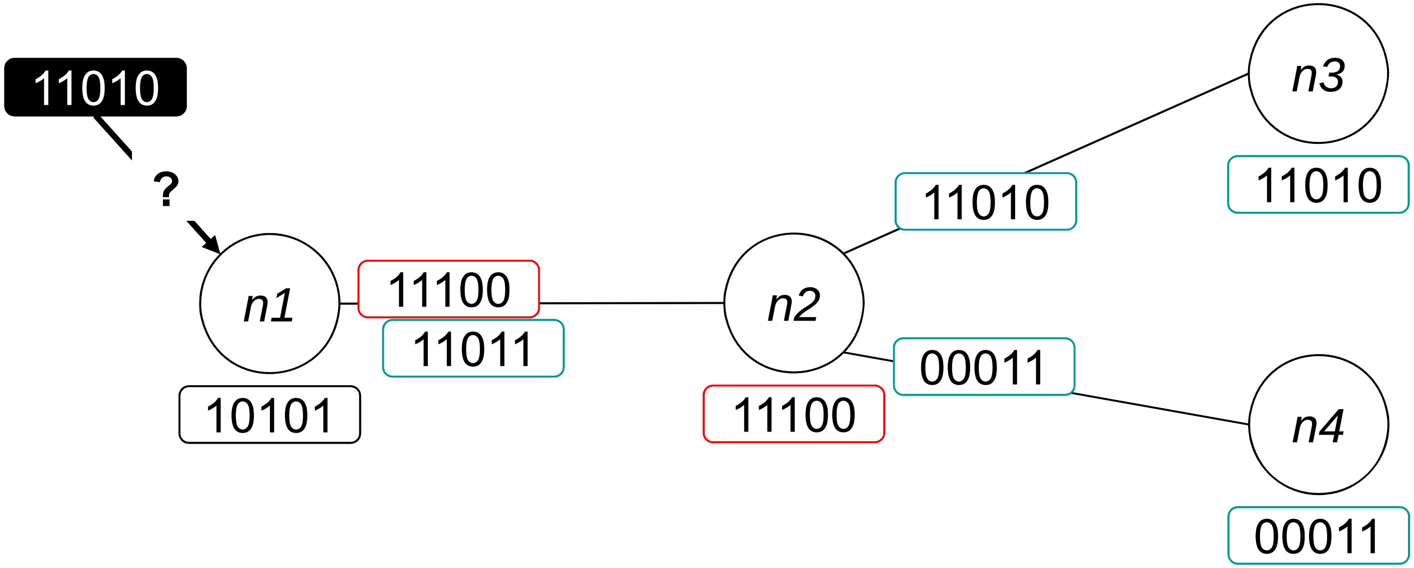


Sean C. Rhea, John Kubiatowicz: Probabilistic Location and Routing; Infocom 2002.

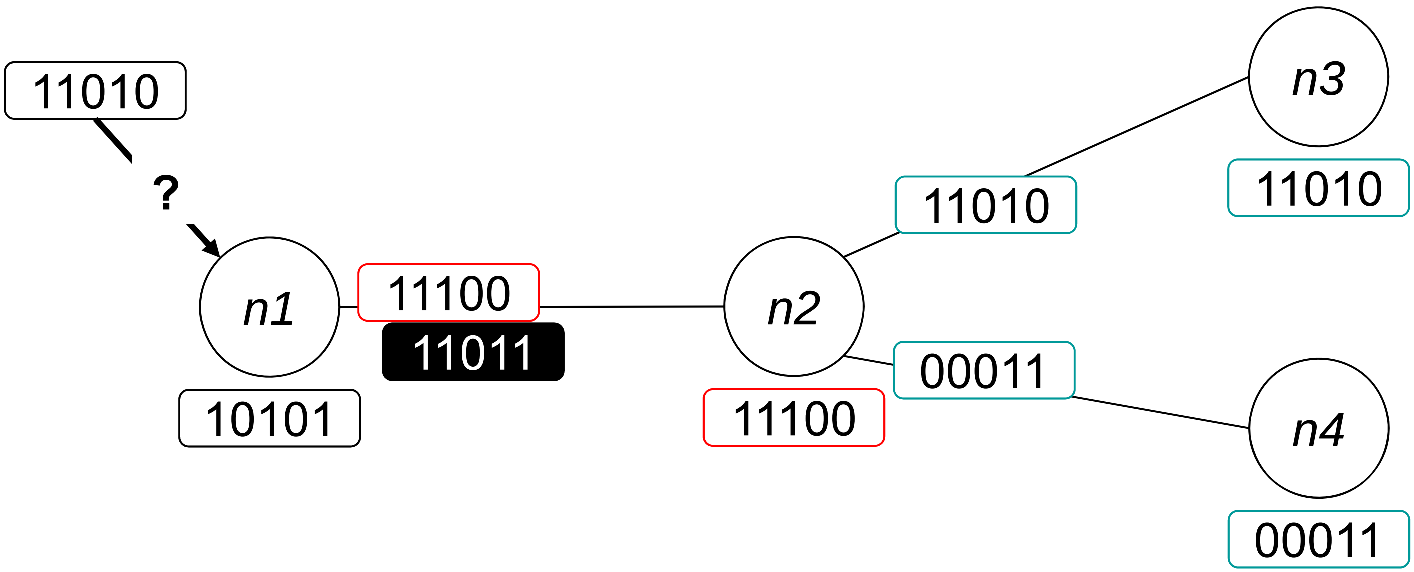
2.2. Example (2)



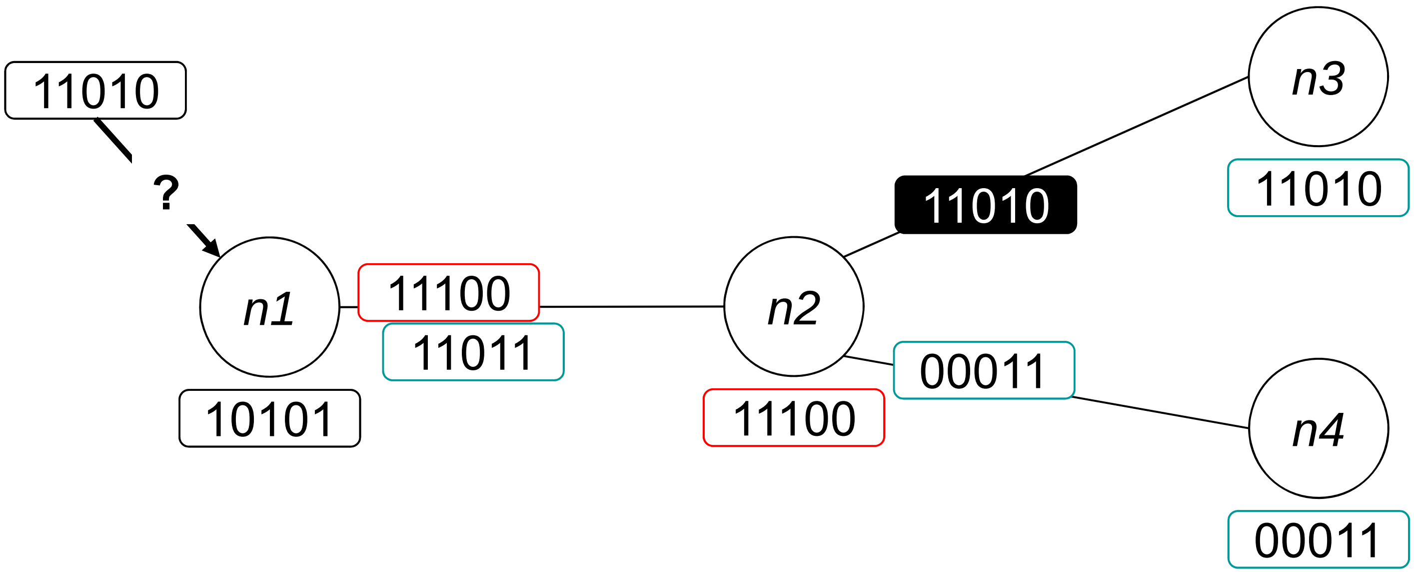
2.2. Query Routing Example



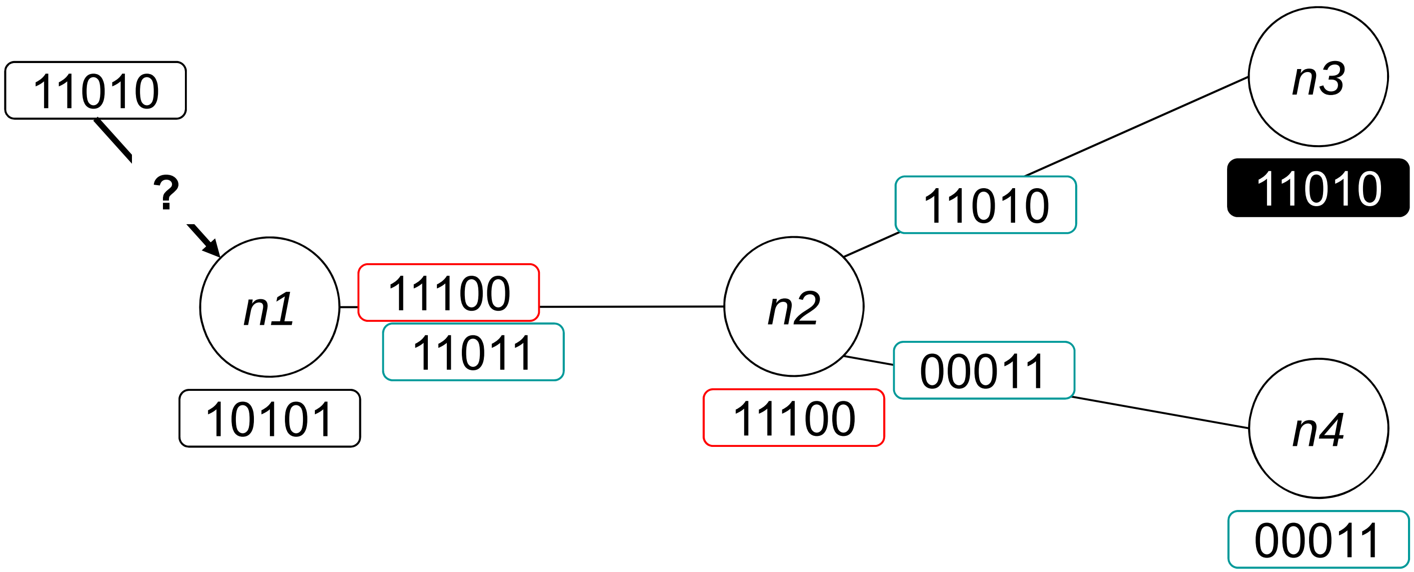
2.2. Query Routing Example



2.2. Query Routing Example



2.2. Query Routing Example



2.3. References

- Traditional Bloom Filter
  - Burton H. Bloom: *Space/Time Trade-offs in Hash Coding with Allowable Errors*; Communications of the ACM, Vol. 13(7), pp. 422-426, 1970.
- Attenuated Bloom Filter
  - S. C. Rhea and J. Kubiatowicz: *Probabilistic Location and Routing*; IEEE INFOCOM 2002, New York, NY, USA, pp. 1248-1257, June 2002.
- Bloomier Filter
  - B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal: *The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables*; SODA 2004.
- Spectral Bloom Filters
  - S. Cohen, Y. Matias: *Spectral Bloom Filters*; SIGMOD 2003.
- Counting Bloom Filters
  - L. Fan, P. Cao, J. Almeida, A. Broder: *Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol*; IEEE/ACM Transactions on Networking, Vol. 8(3), pp. 281-293, June 2000.
- Compressed Bloom Filters
  - M. Mitzenmacher: *Compressed Bloom Filters*; IEEE Transactions on Networking, Vol. 10 (5), pp. 604-612, October 2002.

2.3. References

- A. Broder and M. Mitzenmacher: *Network Applications of Bloom Filters: A Survey*; 40th Annual Allerton Conference on Communication, Control, and Computing, pp. 636-646, 2002.
- T. Kocak, I. Kaya: *Low-Power Bloom Filter Architecture for Deep Packet Inspection*; IEEE Communications Letters, Vol. 10(3), pp. 210-212, March 2006.
- S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood: *Deep Packet Inspection using Parallel Bloom Filters*; IEEE Micro, Vol. 24 (1), pp. 52-61, January 2004.
- R. Laufer, P. Velloso, O. Duarte: *Generalized Bloom Filters*; Technical Report GTA-05-43, COPPE/UFRJ, September 2005.

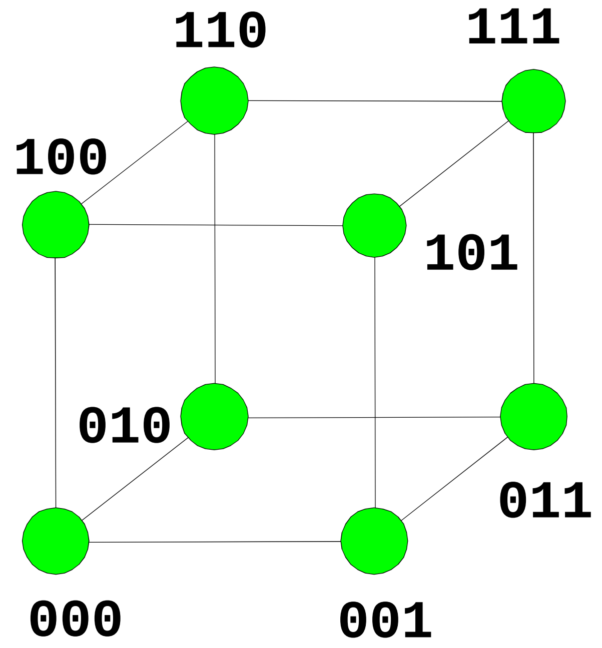
3. Hypercube Networks

Construction, Routing, Properties, Limitations

3. Hypercube Networks

TECHNISCHE UNIVERSITÄT DARMSTADT

- Example:
  - 3-dimensional Hypercube Network

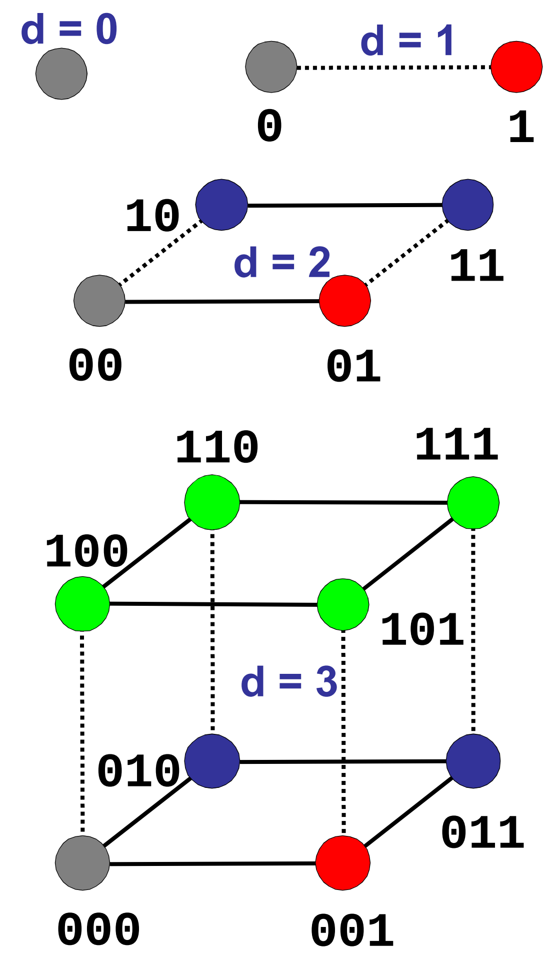


- d-dimensional binary hypercube definition
  - A d-dimensional binary hypercube network consists of 2^d
    - interconnected nodes
    - whose addresses are represented by d-bit binary numbers.
  - A node n has d adjacent nodes
    - whose addresses are obtained by reverting each bit of its address
    - fixed number of contacts per node!

3.1. Hypercube Networks: How to construct

- Construction algorithm
  - if (d == 0)
    draw a node
  - else
    - 1. draw two hypercubes of dimension (d-1)
    - 2. add arcs between
      - corresponding nodes of the two hypercubes

- Identification algorithm
  - 1. use the first bit
    - to decide which of the two sub-hypercubes the node is in.
  - 2. use the remaining d-1 bits as
    - an address within that sub-hypercube.



3.2. Hypercube Networks: Routing Algorithm

TECHNISCHE UNIVERSITÄT DARMSTADT

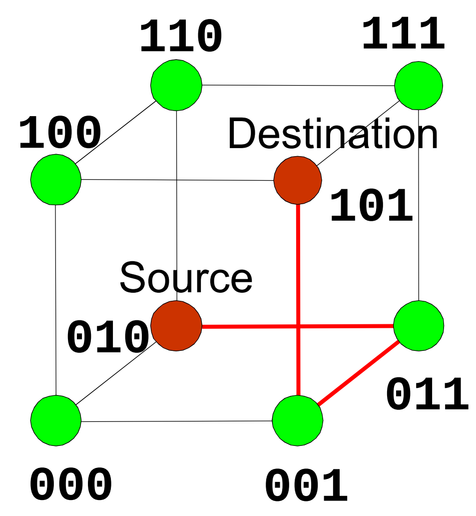
- A simple source-to-destination routing algorithm for binary hypercube networks
  - Node failure is ignored

- To route from node i to node j,
  - the bits of i which are different from those of j are changed,
    - traversing a link with each bit changed,
    - until j is reached.
  - Since each node has links corresponding to each of the n bit positions,
    - any bit can be changed at any step in the route;
  - →
    - the number of eligible links at each step equals the distance from the destination.

3.2. Hypercube Networks: Routing Example

- Route from 010 to 101
  - 010 has 3 options (000, 110, 011)
    - → distance = 3
    - Select 011 as the next hop
  - 011 has 2 options (001, 111)
    - → distance = 2
    - Select 001 as the next hop
  - 001 is direct neighbor
    - → distance = 1
    - Forward to destination

- IF
  - nodes fail
- THEN
  - alternative paths may be followed in dynamic environments



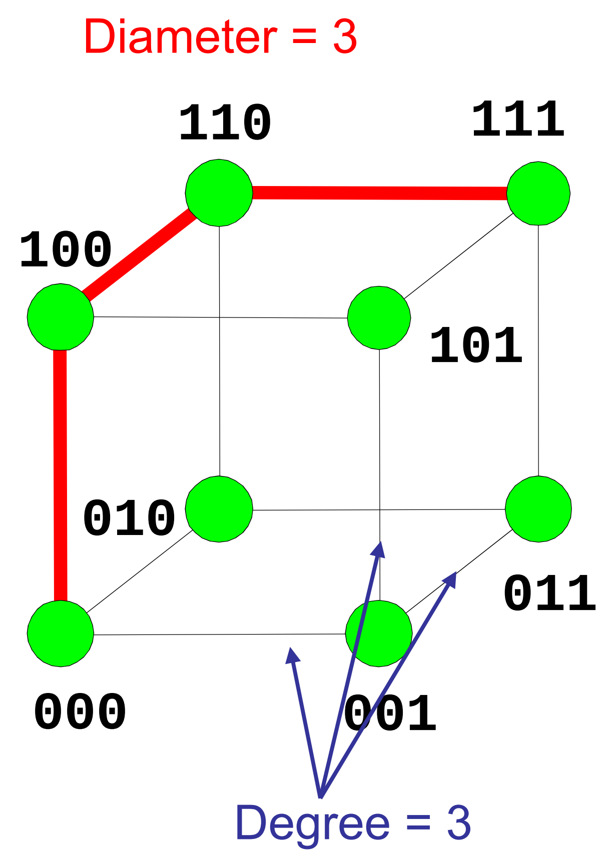
3.3. Hypercube Networks: Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

- Basic properties:

- diameter of the network
  - i.e. worst case node distance
  - increases logarithmically with respect to the network size
- node degree
  - i.e. number of neighbors
  - increases logarithmically with respect to the network size
- network is both
  - vertex- and
  - edge-symmetric graph

- hypercube is a hierarchically recursive network



3.4. Hypercube Networks: Limitations

- Exponentially expandable
  - Network size is defined only for 1, 2, 4, 8, ... nodes
    - for binary hypercubes
  - Not incrementally expandable
    - i.e. network cannot be defined for any arbitrary integer

- Node degree increases logarithmically with respect to the network size
  - → increasing the required maintenance cost

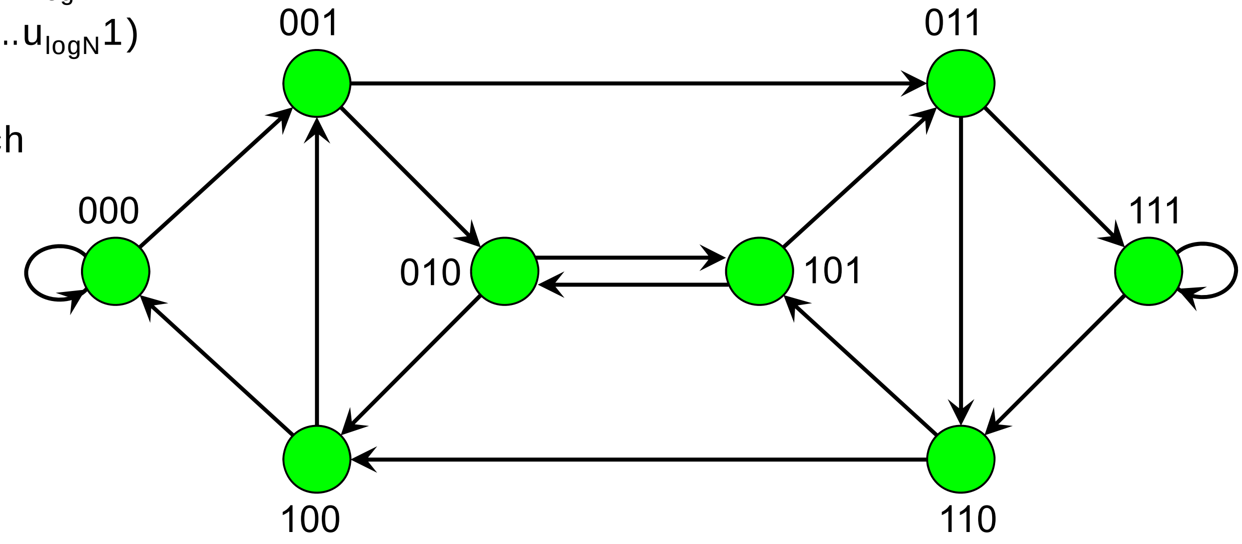
- Mostly appropriate for static environments
  - with infrequent joins and leaves

4. de Bruijn Networks

Construction, Routing, Properties, Limitations, Omicron

4. de Bruijn Networks

- r-dimensional binary de Bruijn digraph (directed graph) consists of
  - 2^r nodes
  - 2^{r+1} edges
- each node corresponds to
  - an r-digit binary string
- there is a directed edge
  - from each node $(u_1 u_2 \ldots u_{\log N})$
  - to $(u_2 \ldots u_{\log N} 0)$ and $(u_2 \ldots u_{\log N} 1)$
  - number of in and out edges per node: 2 each



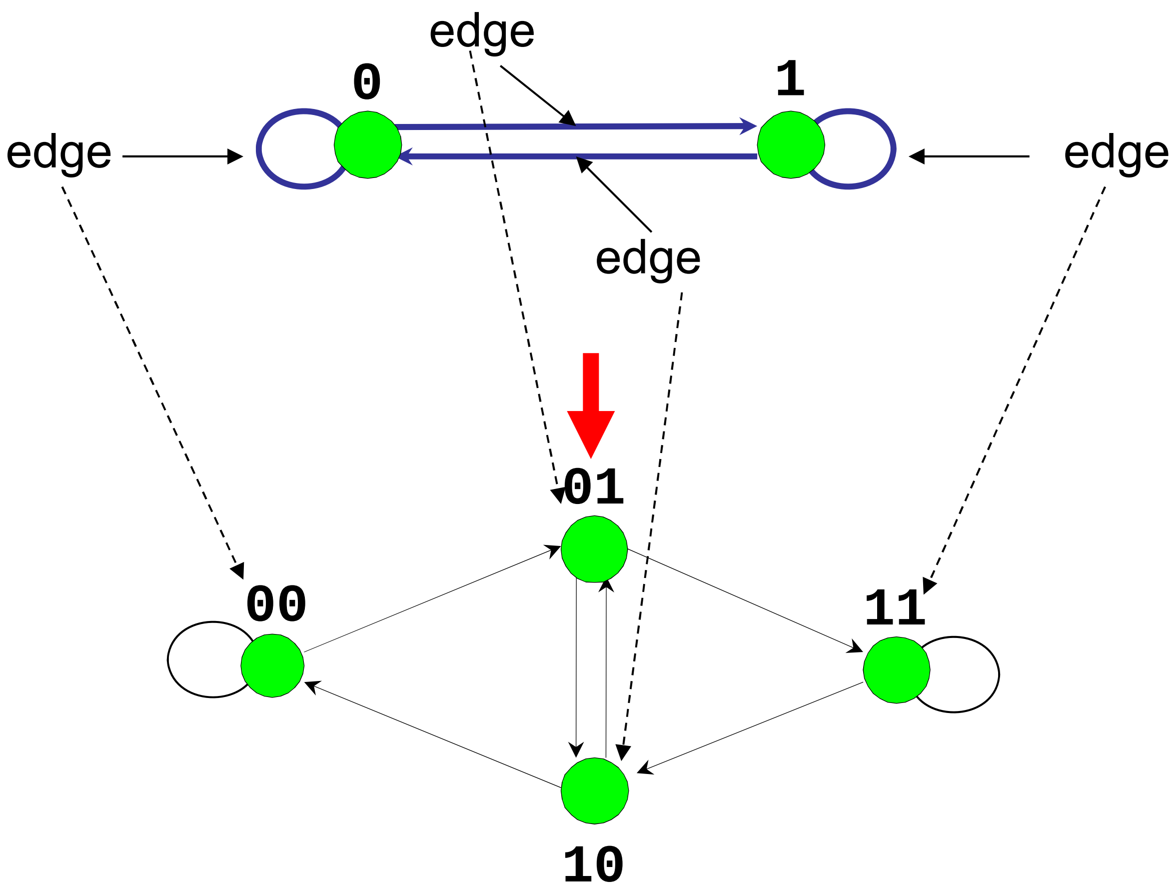
4.1. de Bruijn Networks: How to construct

- Partial Line Digraph Algorithm: recursive construction
  - The N-node graph can be obtained from the N/2 –node graph

- Algorithm description
  - Replace every edge of the N/2-node graph with a node
    - Edge($u_1 \ldots u_{logN-1}$, $u_2 \ldots u_{logN}$) → Node($u_1 \ldots u_{logN}$)
  - Insert a directed edge between pairs of nodes
    that correspond to consecutive directed edges in the N/2-node graph
    - Edge($u_1 \ldots u_{logN-1}$, $u_2 \ldots u_{logN}$) and
      Edge($u_2 \ldots u_{logN}$, $u_3 \ldots u_{logN+1}$)
    - Replaced by
      Edge($u_1 \ldots u_{logN}$, $u_2 \ldots u_{logN+1}$)

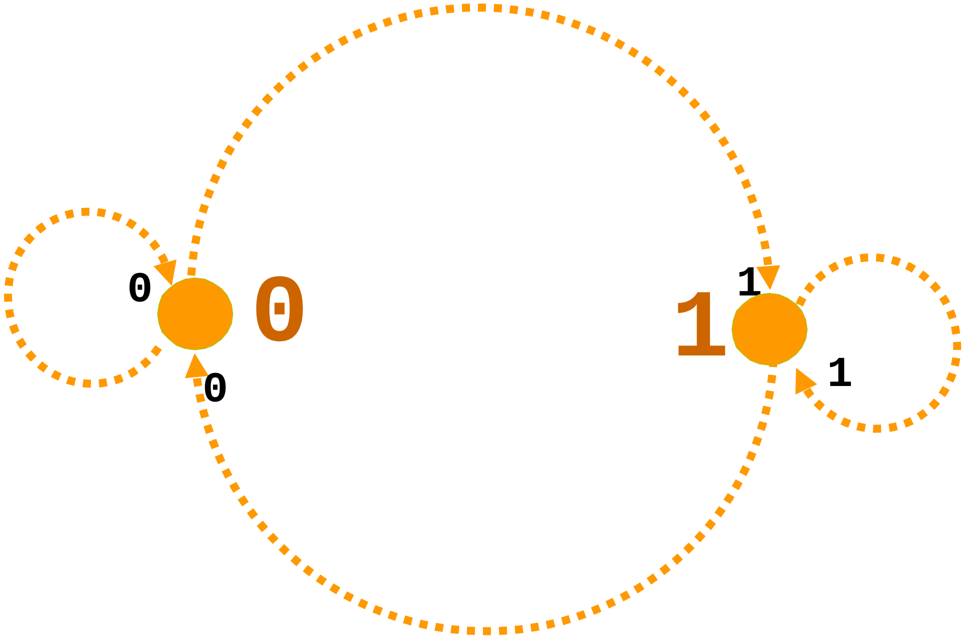
4.1. de Bruijn Networks: Construction Example

- N = 2
- N = 4



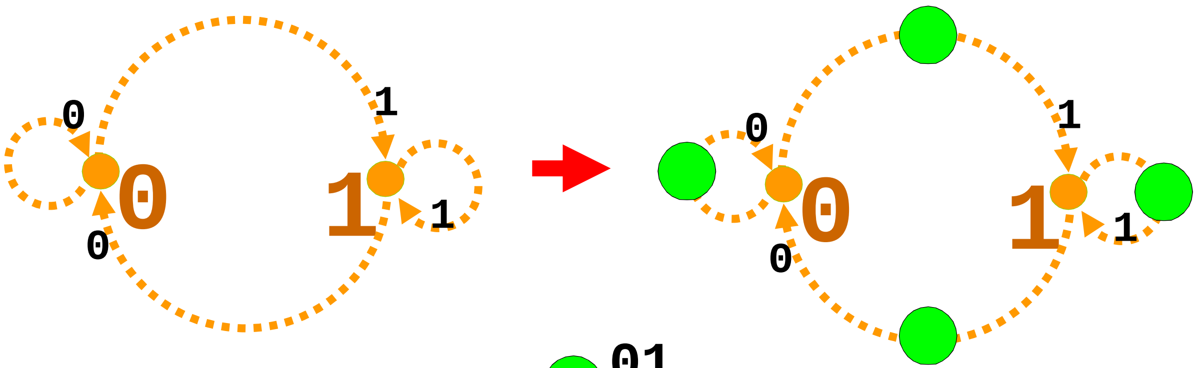
4.1. de Bruijn Networks: Construction Example

- N = 2

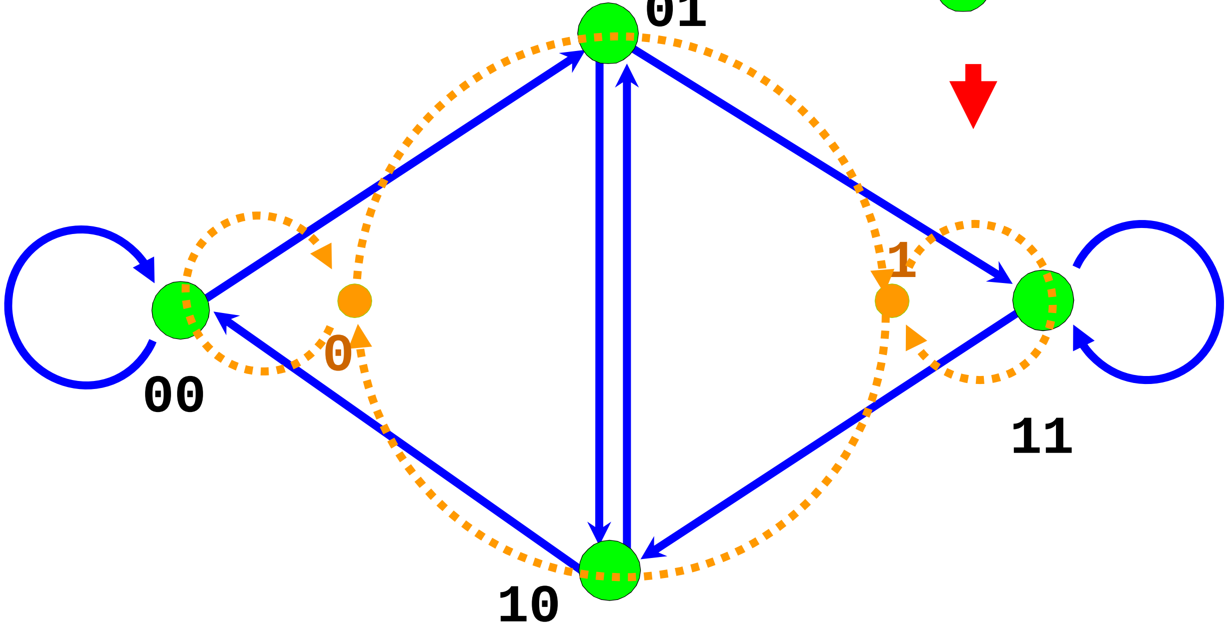


4.1. de Bruijn Networks: Construction Example

- N = 2



- N = 4



4.2. de Bruijn Networks: Routing Algorithm

- Operation: shift_match(shift, K, L), where 0 <= shift <= D
  - returns TRUE if and only if
    $k_{1+shift}k_{2+shift}...k_D = l_1 l_2 ... l_{D-shift}$
  - returns FALSE otherwise
- Operation: merge(shift, K, L), where 0 <= shift <= D.
  - returns the sequence of length (D + shift) given by
    $k_1 k_2 ... k_D l_{D-shift+1} l_{D-shift+2} ... l_D$

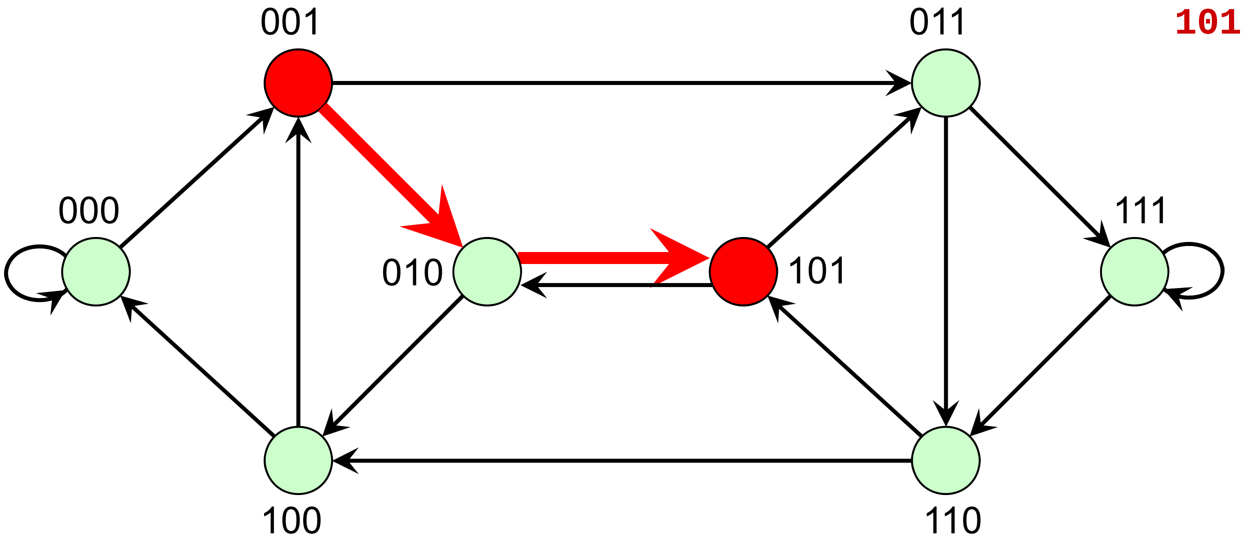
- Shortest-path algorithm (Sivarajan and Ramaswami)

```
shortest_path(K, L)
    shift = 0
    while (shift_match(shift, K, L) == FALSE and shift < D)
     do shift = shift + 1
    return merge(shift, K, L)
```

4.2. de Bruijn Networks: Routing Example

- Example: route
  - to node 101
  - from node with ID (GUID) 001

- Only shift left operation is allowed
  - i.e. only 2 entries in the routing table




```
Example
  to node   101
  from node 001

   ←← match by shift left
  101 destinations node
001       from
 010      via …
  101     to destination node
```

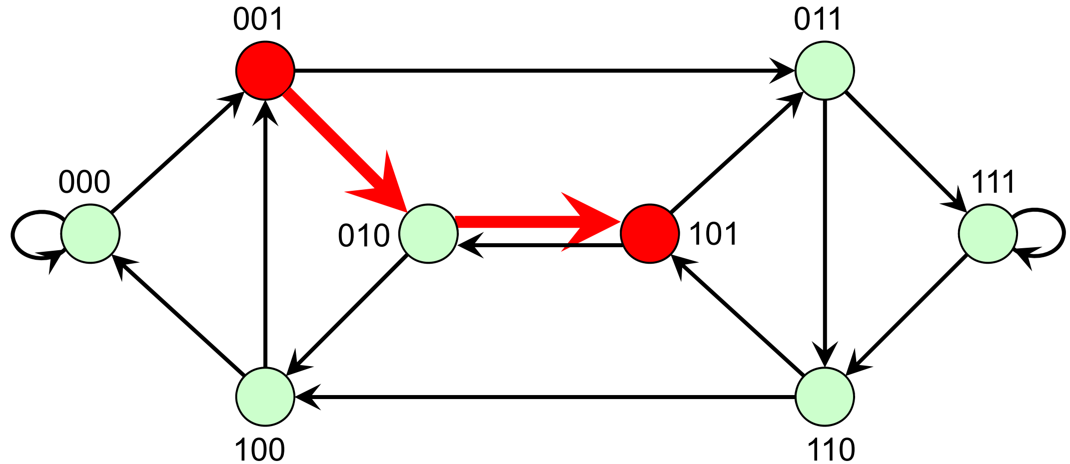
4.2. de Bruijn Networks: Routing Example

- i.e. Route message from source (001) to (101)

```
shift = 0
  shift_match(0, 001, 101) = FALSE  …  001 101
shift = 1
  shift_match(1, 001, 101) = FALSE  …    01   10
shift = 2
  shift_match(2, 001, 101) = TRUE   …     1    1

return merge (2, 001,101) = 00101   …         01
```

  - 2 forwarding steps



4.2. de Bruijn Networks: Routing Example

- We want
  - to route to node with ID (GUID) 101
  - coming from node 001
- Only shift left operation is allowed
  - i.e. only 2 entries in the routing table
- Procedure
  - 1 0 1 destinations node
  - 0 0 1 from
  - 0 1 0 via ...
  - 1 0 1 to destination node

And other example

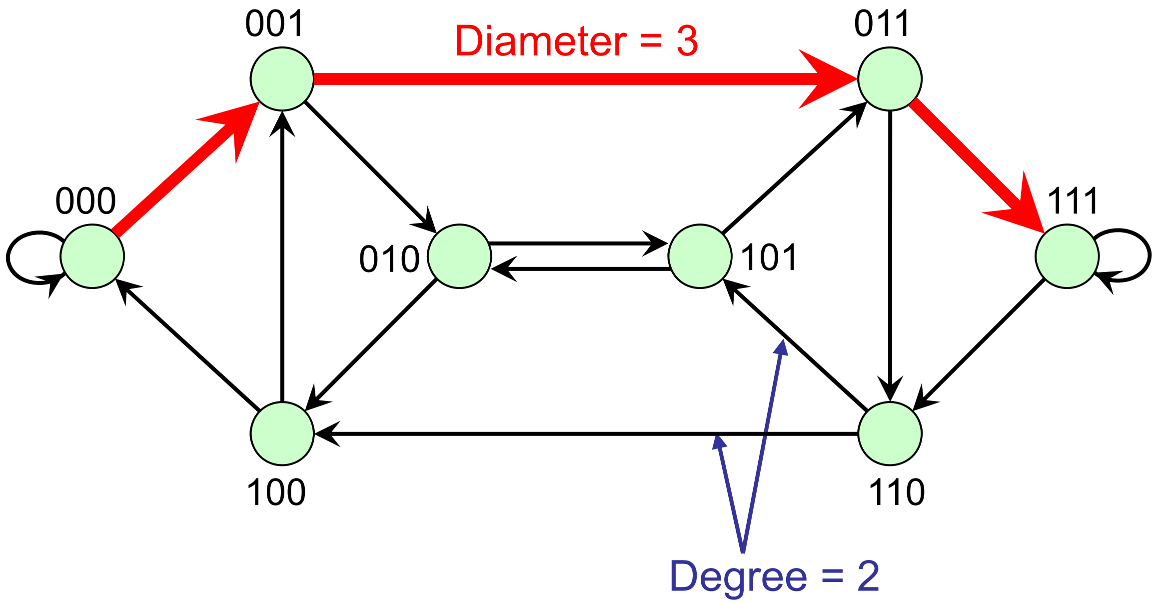
.. to node 01001

.. from 11111

```
     01001 destinations node
11111      from
 11110     via …
  11101
   11010
    10100
     01001 to destination node
```

4.3. de Bruijn Networks: Properties

- Basic characteristics
  - Average distance is very close to the diameter
  - Constant vertex degree
  - Logarithmic diameter
  - Adjacency is based on left shift by 1 position

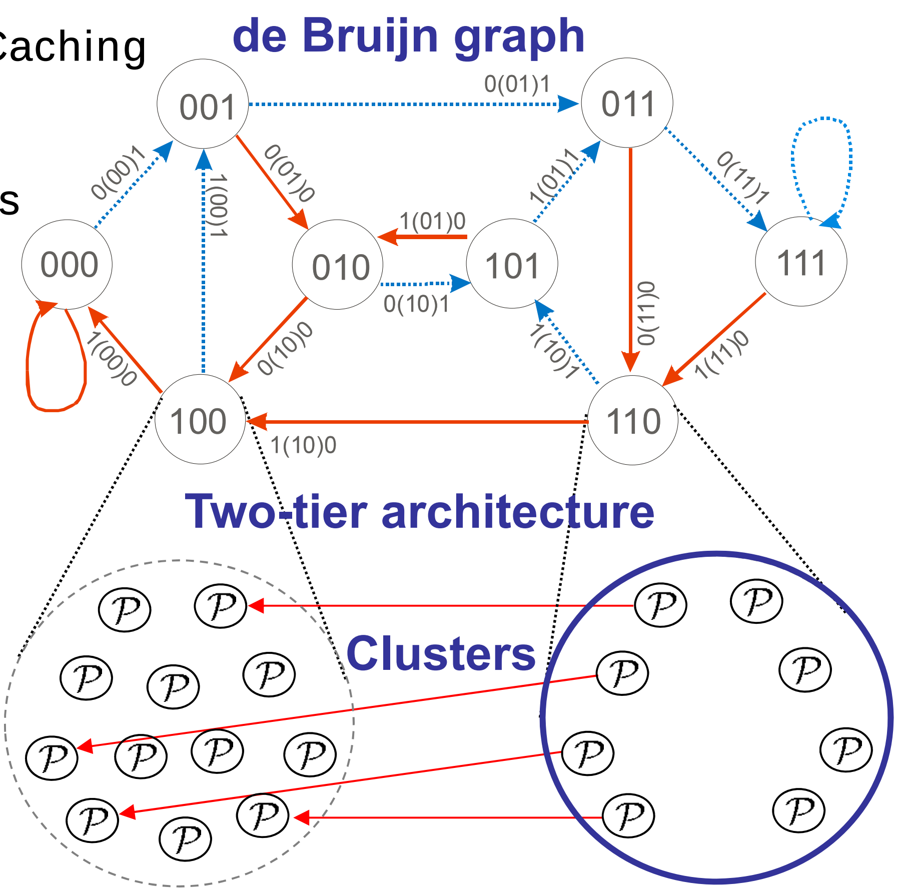


4.4. de Bruijn Networks: Limitations

- Limitations of de Bruijn networks
  - Exponentially expandable
    - Network size is defined only for 1, 2, 4, 8, ... nodes (for binary graphs)
    - Not incrementally expandable
      - i.e. basic network cannot be defined for any arbitrary integer
      - But, enhancements possible (see PhD Darlargiannis)
  - Mostly appropriate for static environments with infrequent joins and leaves

4.5. Omicron

- Omicron
  - Organized Maintenance, Indexing, Caching and Routing for Overlay Networks
  - Uses a hybrid DHT topology to deal with several conflicting requirements
    - Scalability, efficiency, robustness, heterogeneity and load-balance
- Structured macro level (de Bruijn)
  - Scalable
    - Asymptotically optimal Diameter and average node distance
    - Fixed node degree
  - Stable components required
- Clustered micro level
  - Redundancy and fault-tolerance
  - Locality aware
  - Finer load balance
  - Handling hot spots



4.5. Omicron Roles

- Common overlay network operations
  - Ⓜ **Maintainer:** Maintaining structure (topology)
  - Ⓘ **Indexer:** Indexing advertised items
  - Ⓒ **Cacher:** Caching popular items
  - Ⓡ **Router:** Routing queries