A Common API for Structured Peer-to-Peer Overlays

Frank Dabek, Ben Y. Zhao, Peter Druschel, Ion Stoica

Structured Peer-to-Peer Overlay

- They are:
  - Scalable, self-organizing overlay networks
  - Provide routing to location-independent names
  - Examples: CAN, Chord, Pastry, Tapestry, …
- Basic operation:
  - Large sparse namespace *N* (integers: $0–2^{128}$ or $0–2^{160}$)
  - Nodes in overlay network have nodeIds $\in N$
  - Given $k \in N$, a deterministic function maps *k* to its *root* node (a live node in the network)
  - *route(msg, k)* delivers *msg* to *root(k)*

Current Progress

- Lots of applications built on top
  - File systems, archival backup
  - Application level multicast
  - Routing for anonymity, attack resilience
- But do we really understand them?
  - What is the core functionality that applications leverage from them?
  - What are the strengths and weaknesses of each protocol? How can they be exploited by applications?
  - How can we build new protocols customized to our future needs?

Our Goals

- Protocol comparison
  - Compare and contrast protocol semantics
  - Identify basic commonalities
  - Isolate and understand differences
- Towards a common API
  - Easily supportable by old and new protocols
  - Enables application portability between protocols
  - Enables common benchmarks
  - Provides a framework for reusable components

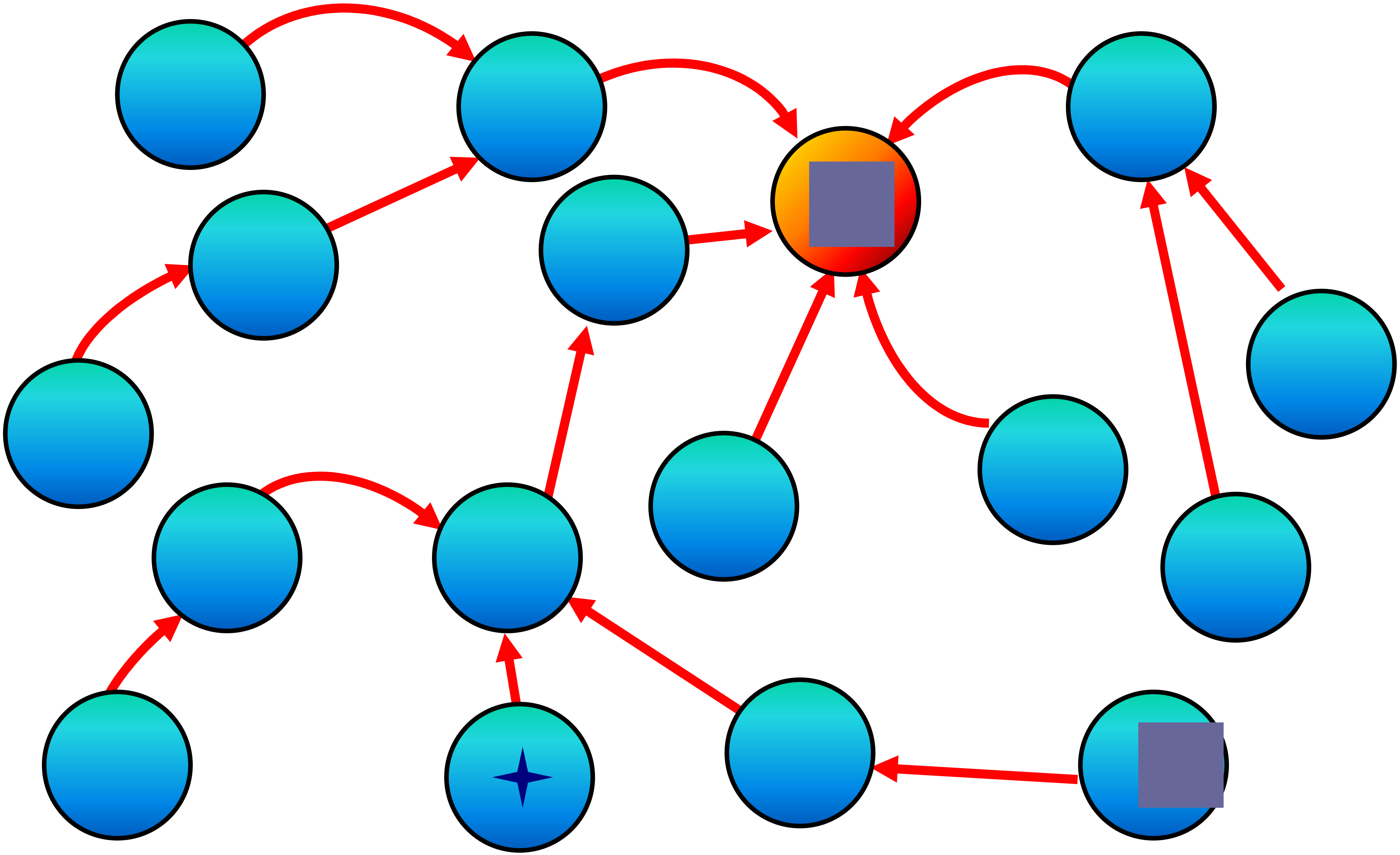
Talk Outline

- Motivation
- DHTs and DOLRs
- A Flexible Routing API
- Usage Examples

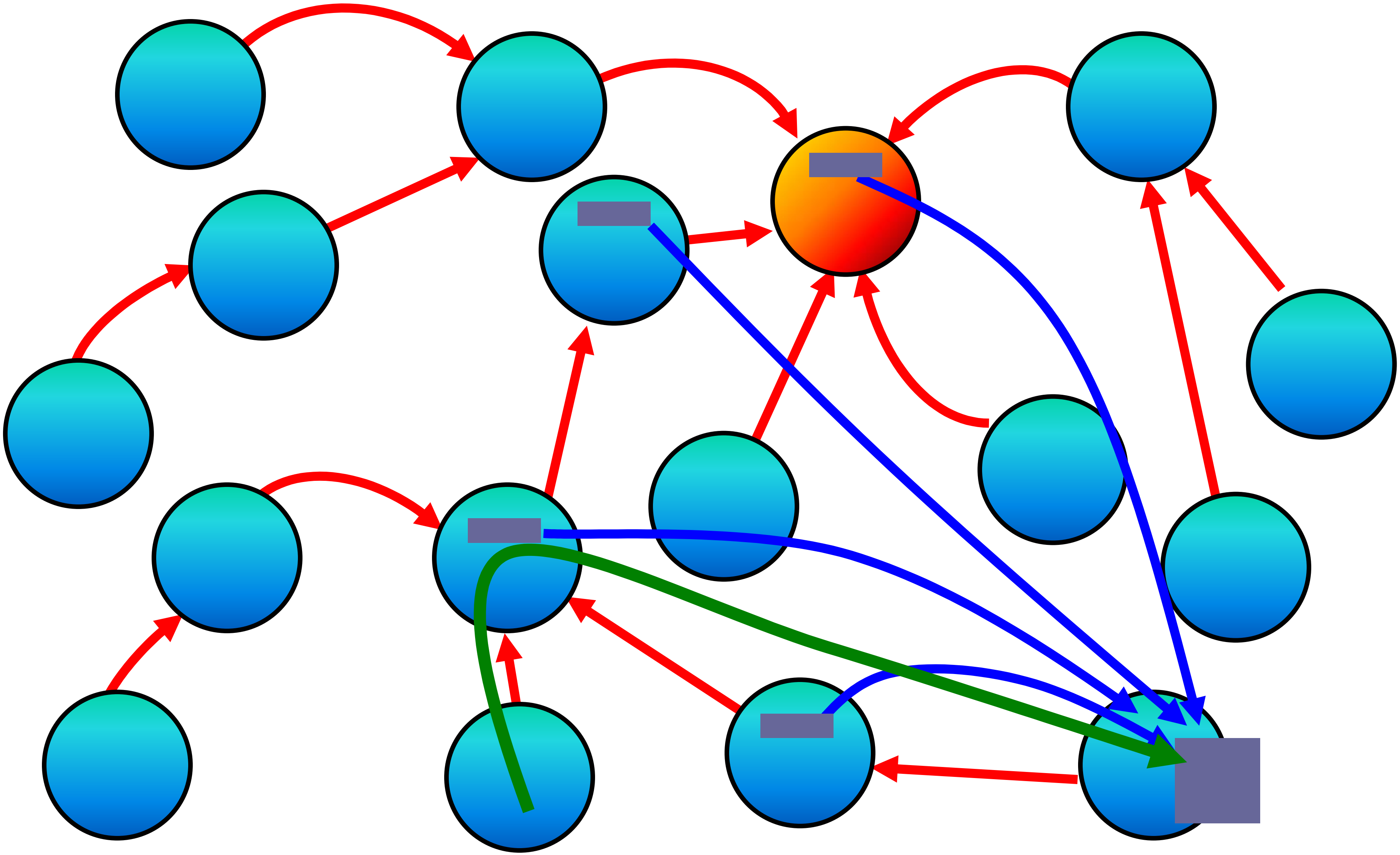
Decomposing Functional Layers

- Distributed Hash Tables (DHT)
  - *put(key, data), value = get(key)*
  - Hashtable layered across network
  - Handles replication; distributes replicas randomly
  - Routes queries towards replicas by name
- Decentralized Object Location and Routing (DOLR)
  - *publish(objectId), route(msg, nodeId), routeObj(msg, objectId, n)*
  - Application controls replication and placement
  - Cache location pointers to replicas; queries quickly intersect pointers and redirect to nearby replica(s)

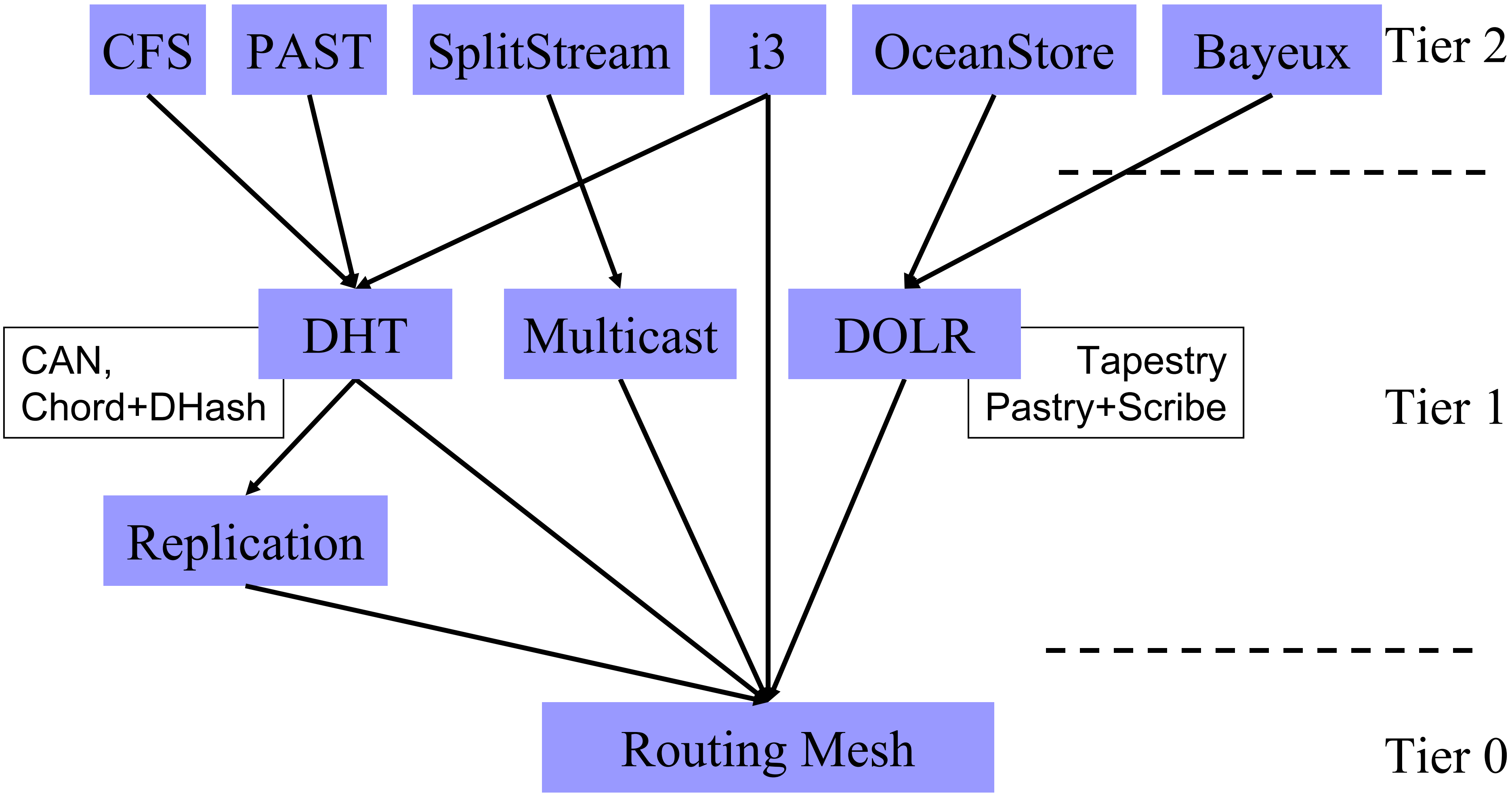
DHT Illustrated



DOLR Illustrated



Architecture



Talk Outline

- Motivation
- DHTs and DOLRs
- A Flexible Routing API
- Usage Examples

Flexible API for Routing

- Goal
  - Consistent API for leveraging routing mesh
  - Flexible enough to build higher abstractions
    - Openness promotes new abstractions
    - Allow competitive selection to determine right abstractions
- Three main components
  - Invoking routing functionality
  - Accessing namespace mapping properties
  - Open, flexible upcall interface

API (routing)

Data types

- Key, nodeId = 160 bit integer
- Node = Address (IP + port #), nodeId
- Msg: application-specific msg of arbitrary size

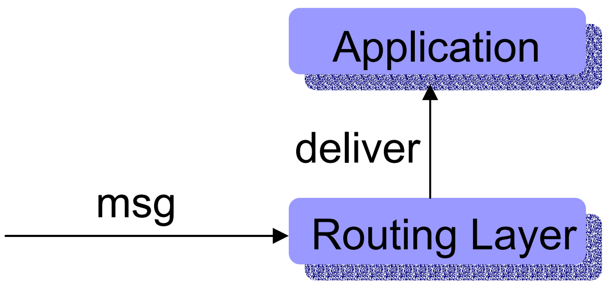
Invoking routing functionality

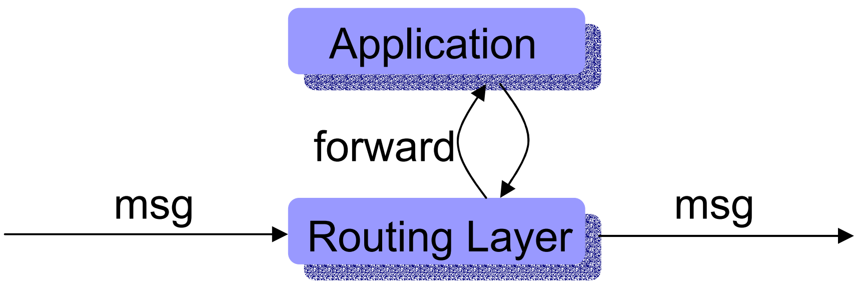
- **Route(key, msg, [node])**
  - route message to node currently responsible for key
  - Non-blocking, best effort – message may be lost or duplicated.
  - node: transport address of the node last associated with key (proposed first hop, optional)

API (namespace properties)

- **nextHopSet = local_lookup(key, num, safe)**
  - Returns a set of at most *num* nodes from the local routing table that are possible next hops towards the *key*.
  - Safe: whether choice of nodes is randomly chosen
- **nodehandle[ ] = neighborSet(max_rank)**
  - Returns unordered set of nodes as neighbors of the current node.
  - Neighbor of rank *i* is responsible for keys on this node should all neighbors of rank < *i* fail
- **nodehandle[ ] = replicaSet(key, num)**
  - Returns ordered set of up to *num* nodes on which replicas of the object with key *key* can be stored.
  - Result is subset of neighborSet plus local node
- **boolean = range(node, rank, lkey, rkey)**
  - Returns whether current node would be responsible for the range specified by *lkey* and *rkey,* should the previous *rank-1* nodes fail.

API (upcalls)





- **Deliver(key, msg)**
  - Delivers an incoming message to the application. One application per node. Demultiplexing done by including demux key in msg.
- **Forward(&key, &msg, &nextHopNode)**
  - Synchronous upcall invoked at each node along route
  - On return, will forward *msg* to *nextHopNode*
  - App may modify *key*, *msg*, *nextHopNode*, or terminate by setting *nextHopNode* to NULL.
- **Update(node, boolean joined)**
  - Upcall invoked to inform app of a change in the local node's neighborSet, either a new node joining or an old node leaving.

Talk Outline

- Motivation
- DHTs and DOLRs
- A Flexible Routing API
- Usage Examples

DHT Implementation

- Interface
  - *put (key, value)*
  - *value = get (key)*
- Implementation (source S, root R)
  - Put: *route(key, [PUT,value,S], NULL)*
    Reply: *route(NULL, [PUT-ACK,key], S)*
  - Get: *route(key, [GET,S], NULL)*
    Reply: *route(NULL, [value,R], S)*

DOLR Implementation

- Interface
  - *RouteNode(msg, nodeId)*
  - *Publish(objectId)*
  - *RouteObj(msg, objectId, n)*
- Implementation (server S, client C, object O)
  - RouteNode: *route(nodeId, msg, NULL)*
  - Publish: *route(objectId, ["publish",O,S], NULL)*
    Upcall: *addLocal([O,S])*
  - RouteObj: *route(nodeId, [n,msg], NULL)*
    Upcall:
    serverSet[] = getLocal(O);
    if (|serverSet|<n), route(nodeId, [n-|serverSet|,msg], NULL)
    for first n entries in serverSet,
    route(serverSet[i], msg, NULL)

Conclusion

- Very much ongoing work
  - Feedback valuable and appreciated
- Yet to come
  - Implementations will move to support routing API
  - Working towards higher level abstractions
    Distributed Hash Table API
    DOLR publish/route API
- For more information, see IPTPS 2003

- Thank you…