P-Chord: Formally Verified Distributed Hash Table using P

CS3551: Advanced Topics in Distributed Information Systems

Derrick Hicks Shinwoo Kim
{ddh32,shinwookim}@pitt.edu

Fall 2024

The LOOKUP Problem

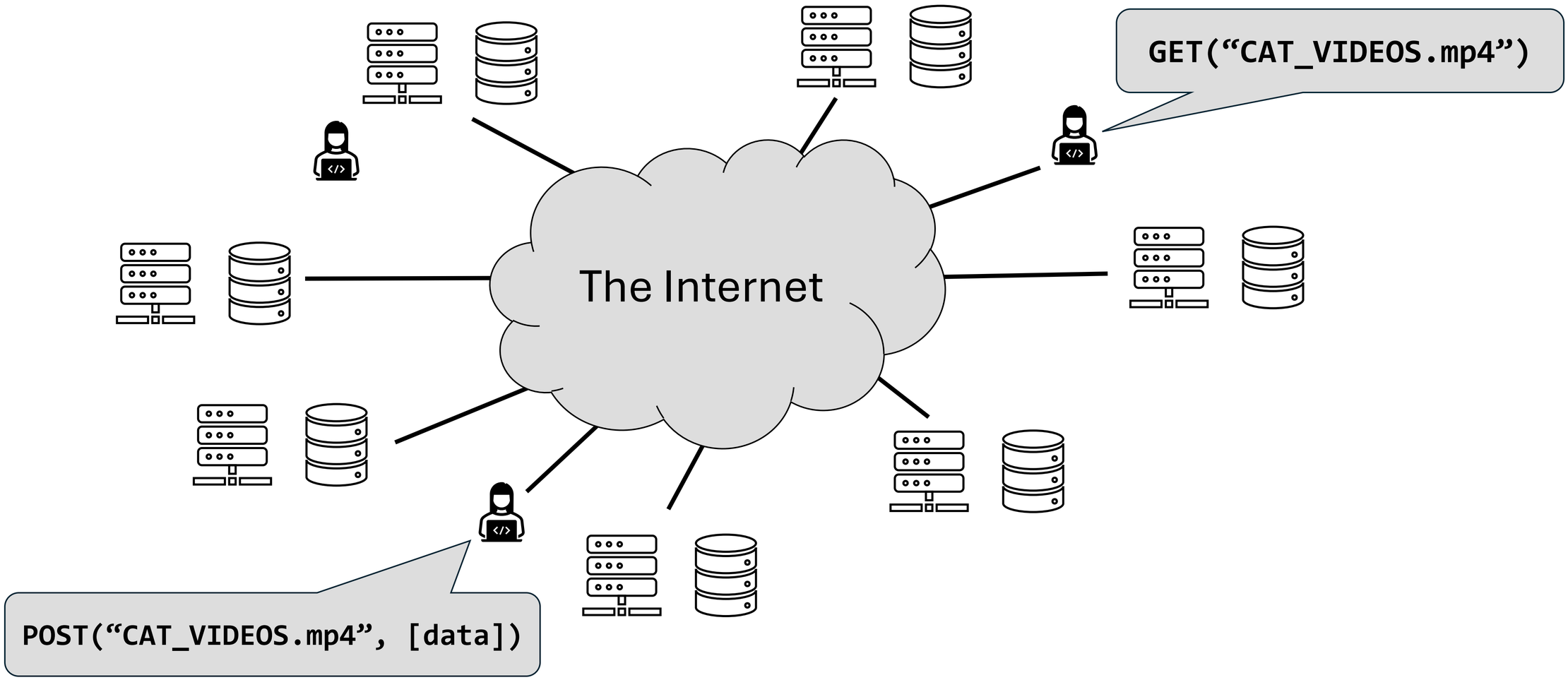


Figure Credit: Wyatt Lloyd’s COS 418 Lecture Slides (Princeton)

Hash Tables

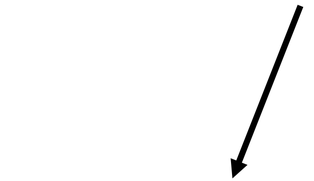
- Hash Tables are essential building blocks in modern software systems
  - Python's built-in dictionary(dict)
  - C++'s map(std::map)[1]
  - Everything in JavaScript except the "seven primitives"
- Provides *O(1)* look-up[2]
- Idea: Expand this concept to be internet-scale

1. Technically, `std::map` is a Balanced Search Tree; but the `std::unordered_map` is based on a Hash Table
2. Assuming an ideal deterministic hash function *f*

Distributed Hash Table (DHT)

- An abstraction of *hash table* in a distributed setting
  - Use cases: Peer-to-peer file-sharing, Large-scale storage management, Web Mirroring …
  - Benefits: Decentralized, Scalable, Efficient, and Dynamic

```
key = hash(data)
IP_addr = dht.lookup(key)
POST(IP_addr, key, data)
…
data = GET(IP_addr, key)
```



Nodes can **fail**, new nodes can join

However, Implementing correct distributed protocols is notoriously difficult.

Abstract

The Paxos algorithm, when presented in plain English, is ver[illegible]ple.

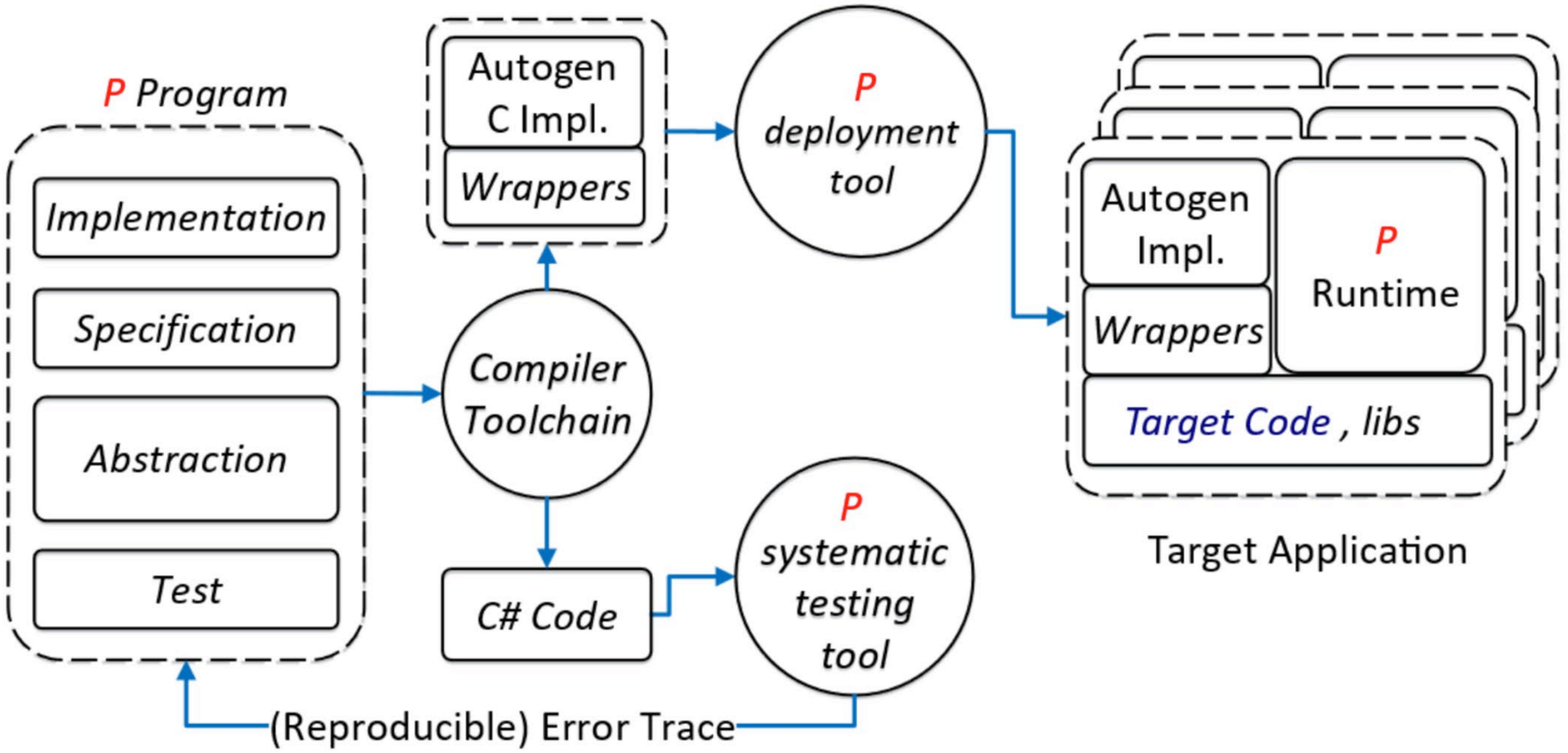
- Formal verification attempts to address this issue
- But critics have said that verification can easily become detached from the implementation
  - And cause too much overhead

P Programming Language

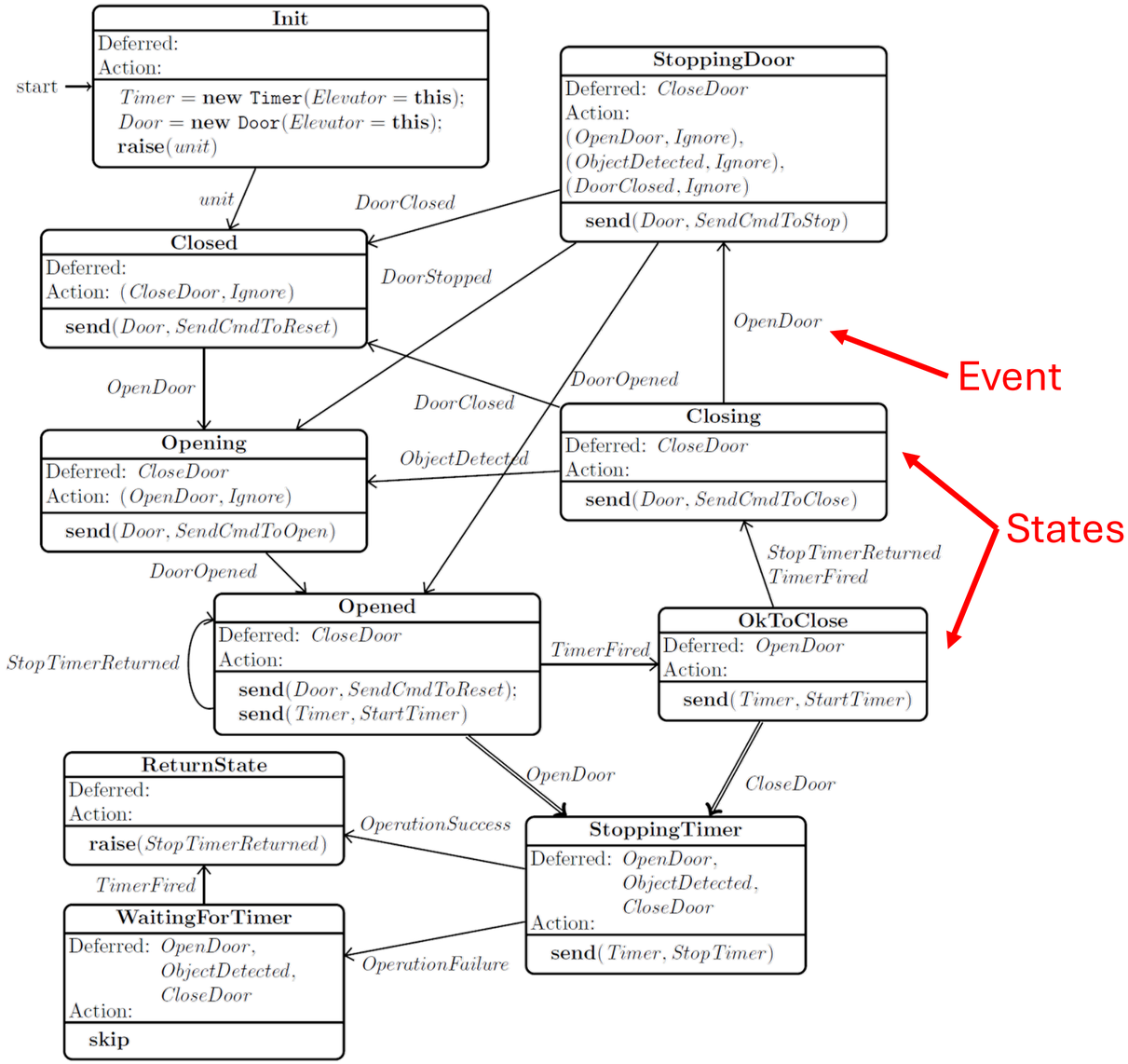
- Traditional approach to developing a reactive system:
  - **Implementation** in Java, C#, C.
  - **Analysis** by modeling in formal framework like Maude, K, Coq.
- P's approach: **unified framework** for both implementation and modeling
  - Systems (and environment) as state machines interacting via events
  - Built-in model checking for proving properties
  - *Generates executable C# code that can integrate with other software*

- Used by Microsoft in Windows 8 USB stack

P Programming Language



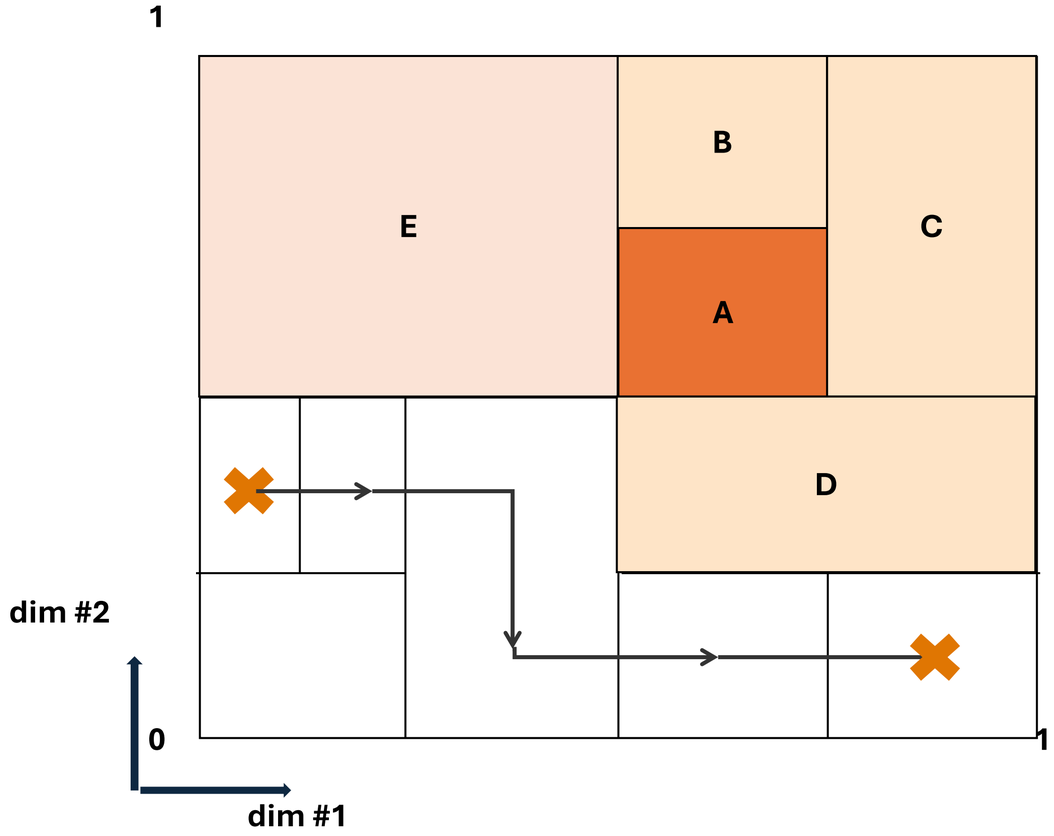
Systems as state machines



Systems as state machines

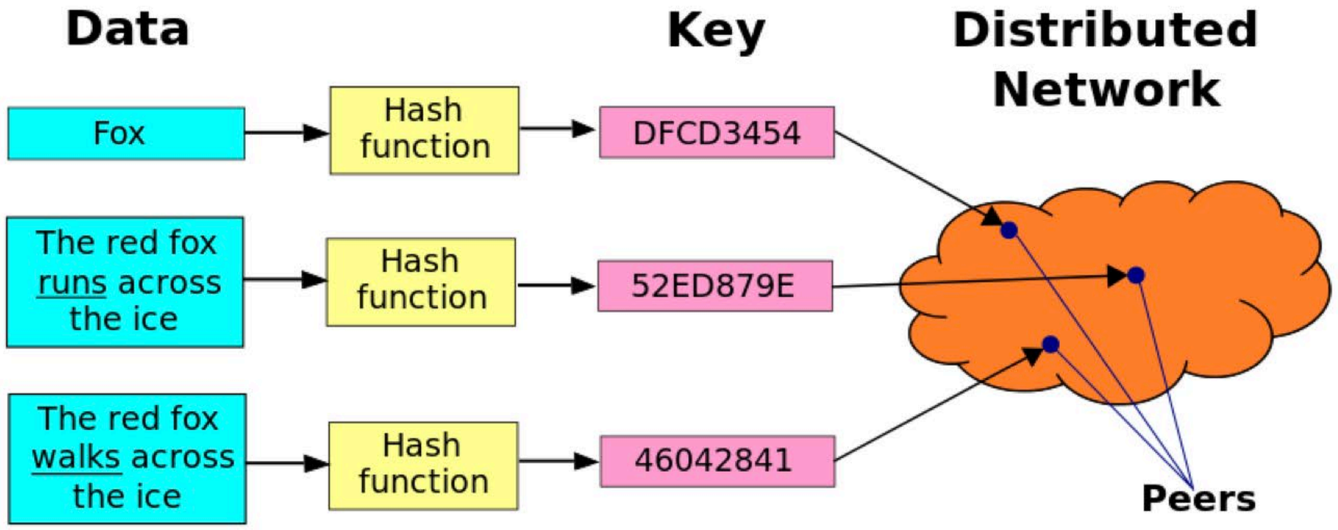
Original Project Goals

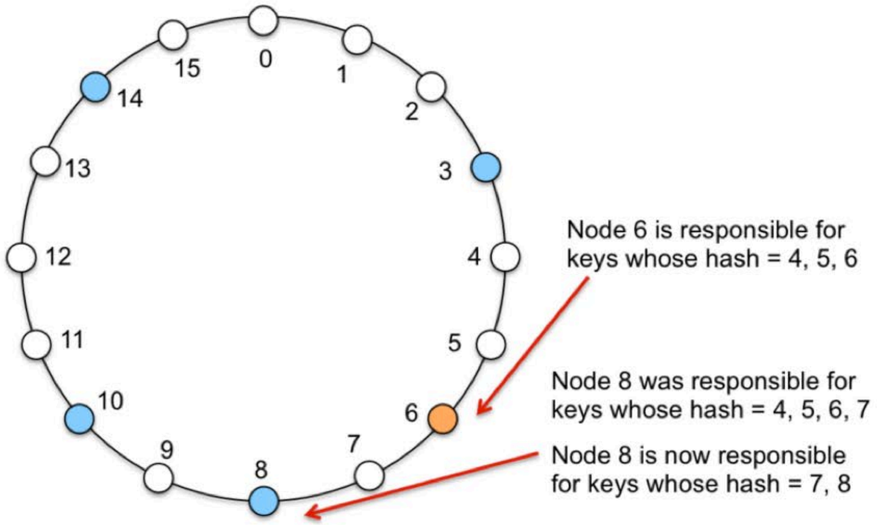
- Goal: Implement *Content Addressable Network (CAN) using the P Language*



Revised Project Goals

- Goal: Implement *Chord* *using the P Language*





Revised Project Goals

- Goal: Implement *Chord using the P Language*
  - From "*Chord: A scalable peer-to-peer lookup service for Internet applications*" [SIGCOMM 2001]

- Why?
  - Not much recent work on content addressable networks, but chord is still seeing interest
    - Received SIGCOMM Test-of-Time Award
    - Some from researchers working in formal verification
      - "*Reasoning about Identifier Spaces: How to Make Chord Correct*" [IEEE 2017]
      - "*Verification of the Chord protocol with TLA+*" [UiT Thesis]
  - A lot simpler to construct correctness criterions
    - Although there are still challenges

Constructing Chord Protocol

Chord Design

- The Chord protocol organizes nodes into a structured ring topology, where each node is responsible for managing a subset of keys, up to and including its own identifier.
- Every node maintains a successor, which is responsible for managing the keys in the interval (n,successor].
  - A lookup request is forwarded through the ring to successive nodes until it reaches the node responsible for the target key. In the worst case, this process requires O(n) hops.
  - To optimize lookup efficiency, Chord typically employs a finger table with m entries, enabling logarithmic lookup times for a network with up to 2^m nodes. However, this optimization will not be implemented in our project

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

Chord Design

- Nodes can dynamically join or leave the network.
  - When a node leaves, it transfers its keys to its successor node.
  - Conversely, when a node joins, it retrieves the keys within its assigned interval from its successor.
- As nodes join, the successor may need to be adjusted to ensure that each nodes acts as the successor for one node only.
  - Each node maintains a reference to its predecessor, and periodically poll their successor's predecessor to determine if a better successor exists
  - Additionally, nodes will notify their successor to confirm they are a potential predecessor.

Credit: Jørgen Aarmo Lund’s thesis on “Verification of the Chord protocol with TLA+”

Fault-tolerant Chord

- In real-world scenarios, servers are susceptible to failure, and if a node cannot communicate with its successor, then a node may no longer be able to forward queries to other nodes.
- To combat this, each node tracks the next *r* successor nodes, whenever a node can no longer communicate with its immediate successor, it removes that node from its list of successors.

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

P Program - Init

```
machine Node {
    var Id: int;
    // Number of nodes in Chord
    var N: int;
    // Next consecutive successors for fault-tolerance
    var succesorList: seq[Node];
    // Size of our successorList
    var R: int;
    // Preceeding node for this node
    var predecessor: Node;
    // key-value pairs
    var Keys: map[int, int];
    // Timer for timeouts to detect failures
    var timer: Timer;

    // Needs to be modified to account for creation of a chord w/
    // a single node and a new node joining the network
    start state Init {
        defer eShutDown, eFindSuccessorReq, eLeave, eStabilize, eNotify;

        entry (n; int, r: int) {
            N = n;
            R = r;
        }
```

```
    on eConfig do (metadata: tNodeConfig) {
        // Determine who is our next R successors
        InitializeSuccessorList(metadata);
        // Initialize the predecessor
        var idx: int;
        var i: int;
        while (i < sizeof(metadata.nodeIds)) {
            if(Id == metadata.nodeIds[i]) {
                idx = i;
                break;
            }
        }
        i = (idx + sizeof(metadata.nodesIds) - 1) % sizeof(metadata.nodesIds)
        predecessor = nodes[i];
        // Wait for incoming requests
        goto WaitForRequests;
    }

    on eJoin do (metadata: tNodeConfig) {
        // Determine who is our next R successors
        InitializeSuccessorList(metadata);
        // Notify successor that we exist and are their predecessor
        send succesorList[0], eNotifySuccesor, this;
        // Wait for incoming requests
        goto WaitForRequests;
    }
}
```

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

P Program - WaitingForRequests

```
state WaitForRequests {

    on eFindSuccessorReq do (node: Node) {
        var successor: Node;

        succesor = succesorList[0];
        // Find the next successsor closest to the given id
        if(InBetween(node.Id, Id + 1, succesor.Id)) {
            send node, eFindSuccessorResp, succesor;
        } else {
            send succesor, eFindSuccessorReq, node;
        }

    }
```

```
on eFixSuccessorList do {
    // Remove immediate successor
    var i: int;
    var succesor: Node;
    // Ping the next successors to see who is live
    while (i < sizeof(succesorList)) {
        succesor = succesorList[0];
        send succesor, eSuccessorListReq, this;
        receive {
            case eSuccessorListResp: (list: seq[Node]) {
                if(sizeof(list) == 0) { continue; }
                succesorList = list;
                succesorList -= (sizeof(list) - 1);
                succesorList += (0, succesor);
                break;
            }
            case eNodeUnavailable: {
                succesorList -= (0);
                continue;
            }
        }
    }
}
```

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

P Program - Stabilize & Notify

```
// Randomly chosen to check if node is in ideal state
on eStabilize do {
    var newSuccesor: Node;
    // Obtain predecessor of our current successor
    send succesorList[0], ePredecessorReq, this;
    goto WaitForGetPedecessorResp;
}

// Possibly have new predecessor
on eNotifySuccesor do (node: Node) {
    // If our predecessor does not exist (default(machine)) or the node that pingd us has
    // a lesser id than our current predecessor
    if(predecessor == default(machine) || InBetween(node.nodeId, predecessor.nodeId, nodeId)) {
        predecessor = node;
    }
}
```

```
state WaitForGetPedecessorResp {
    defer eSuccessorListReq;

    on eGetPredecessorResp do (node: Node) {
        // Verify if a more ideal successor exists
        if(node != default(machine) && InBetween(node.Id, nodeId + 1, succesorList[0].Id - 1)) {
            succesorList -= (0);
            succesorList += (0, node);
        }
        // Ping our new successor to know we exist
        send succesorList[0], eNotifySuccesor, this;
        goto WaitForRequests;
    }

    on eNodeUnavailable do () {
        succesorList -= (0);
        goto WaitForRequests;
    }
}
```

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

Constructing Correctness Properties

Chord Requirements

- The original Chord paper makes many claims about the protocol and its guarantees
  - Some of them have been questioned (at least in its original iteration)
    - *See Using Lightweight Modeling to Understand Chord [IEEE 2013]*
  - Some are not necessarily relevant to correctness
    - E.g., Probabilistic and quantitative claims regarding space-time complexity

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

Chord Requirements

- **Type Safety**
  - At no point should the protocol permit nodes to have successors linking to invalid destinations.
- **Deadlock Freedom**
  - There should always be at least one node so that the protocol can proceed.
- **Network Safety**
  - The protocol should not allow actions that partition the network or leave it in a state where it cannot become ideal.
- **Network Liveness**
  - After all nodes have finished joining, the protocol should eventually have arranged the network into an ideal state.

Credit: Jørgen Aarmo Lund's thesis on "Verification of the Chord protocol with TLA+"

Correctness Properties

(AtLeastOneRing). There must be a ring, which means there must be a non-empty set of ring members.

(AtMostOneRing). There must be no more than one ring, which means that from each ring member, every other ring member is reachable by following the chain of successors.

(OrderedRing). On the unique ring, the nodes must be in identifier order.

P Spec - AtLeastOneRing

```
/**********************************
We would like to assert the AtLeastOneRing property that:
There must be a ring, which means there must be a non-empty set of ring members.
**********************************/
spec AtLeastOneRing observes eVerify
{
    var nodes: set[Node];

    start state Init {
        defer eJoin, eLeave, eShutDown;
        on eMonitor_AtLeastOneRing goto WaitForEvents;
    }
```

```
    state WaitForEvents {
        on eVerify do {
            assert sizeof(nodes) > 0;

            var visited: set[Node];
            var startNode: Node;
            var currNode: Node;
            var i: int;

            // Choose a random starting node
            startNode = choose(nodes);
            currNode = startNode.succesorList[0];

            while (i < sizeof(noddes)) {
                // Cycle already detected -> one ring exists
                if(currNode in visited) {
                    break;
                }
                // Add current node to visited nodes
                visited += currNode;
                // Go to successor
                currNode = currNode.succesor[0];
            }

            if(i == sizeof(nodes)){
                assert currNode.Id == startNode.Id;
            }
        }
    }
```

Lessons Learned

- Formal verification requires thinking about systems as state machines which may not be intuitive
  - Unlike TLA/other approaches, both the *spec* and the *executable code* are written in P
    - So, the system needs to be thought of in terms of state machines and their interactions from an implementation perspective
- Coming up with a precise correctness criterion is hard
  - Too narrow: doesn't capture the correctness requirements
  - Too wide: hard to prove & guarantee
- "P" is not a good programming language name
  - Terrible SEO, impossible to search for information on-line
- (Some) researchers are not good at documenting code
  - P documentation is "perpetually out-of-date" [HackerNews]