Peer to Peer Systems and Blockchains - Midterm

Leonardo Vona
545042

April 14, 2021

1 Main features of CAN

A CAN (Content-Addressable Network) is a distributed hash table whose overlay network represents a virtual d-dimensional Cartesian coordinate space on a d-torus, which is dynamically partitioned among all the nodes in the system. Two nodes are defined *neighbors* if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension.

A key is mapped onto a point in a zone, and the relative value is stored into the node owning the zone. The request may be routed through the CAN infrastructure until it reaches the node in whose zone the point lies. A simple routing strategy works by following the straight line path through the Cartesian space from source to destination coordinates. A better routing metric exploits the network-level round-trip-time (RTT), where for a given destination, a message is forwarded to the neighbor with the maximum ratio of progress to RTT.

For a d-dimensional space partitioned into n equal zones the average routing path length is $(d/4)(n^{1/d})$ and individual nodes maintain $2d$ neighbors.

Many different paths exist between two points in the space and so, even if one or more of a node's neighbors were to crash, a node would automatically route along the next best available path. In case of a node loses all its neighbors in a certain direction, and the repair mechanisms have not yet rebuilt the void in the coordinate space, a node may use an expanding ring search to locate a node that is closer to the destination itself.

A new node that joins the system must be allocated its own portion of the coordinate space. This is done by an existing node splitting its allocated zone in half, retaining half and handing the other half to the new node. In a more uniform partitioning, the target node, instead of directly splitting its own zone, first may compare the volume of its zone with those of its immediate neighbors in the coordinate space, choosing the zone with the largest volume.

When a node wants to leave, it explicitly hand over its zone and associated (key, value) pairs to one of its neighbors. Node failures are handled through an immediate takeover algorithm that ensures one of the failed node's neighbors takes over the zone. The normal leaving procedure and the immediate takeover algorithm can result in a node holding more than one zone. To prevent repeated further fragmentation of the space, a background zone-reassignment algorithm runs to ensure that the CAN tends back towards one zone per node.

Some techniques can be used to improve the performance of a CAN:

- Multi-dimensioned coordinate spaces: increase the dimensions of the CAN coordinate space, reducing the routing path length and improving the fault tolerance, for a small increase in the size of the coordinate routing table.
- Multi coordinate spaces: maintain multiple, independent coordinate spaces, called *realities*, with each node in the system being assigned a different zone in each coordinate space, improving data availability, fault tolerance and reducing path latency.
- Overloading coordinate zones: allow multiple nodes to share the same zone. With zone overloading, a node maintains a list of its *peers* (nodes in the same zone) in addition to its neighbors list, which is one for each neighboring zones. This technique reduces path length, per-hop latency and improves fault tolerance at the cost of an higher system complexity.
- Multiple hash functions: use k different hash functions to map a single key onto k points in the coordinate space, improving data availability.
- Topologically-sensitive construction of the CAN overlay network: use construction mechanisms that take into account the topological proximity of nodes on the underlying IP network to reduce the latency.
- Caching and replication techniques for "hot spot management": use caching and replication techniques to make popular data keys widely available, achieving an higher load balancing.

2 Node joining

The process of node joining is composed by three steps: Bootstrap, Finding a Zone and Joining the Routing. The phases are exemplified in a 2-d space CAN instance.

Bootstrap In order to join the CAN, a new node first discovers the IP address of any node currently in the system through a CAN bootstrap node. This is done with the assumption that a CAN has an associated domain name, and that this resolves to the IP address of one or more CAN bootstrap nodes. A new node then looks up the CAN domain name in DNS to retrieve a bootstrap node's IP address. The bootstrap node supplies the IP addresses of several randomly chosen nodes supposed to be currently in the system. Figure 1 shows an example interaction between the new node and a bootstrap node retrieved via a previous DNS lookup. The new node sends a request to the bootstrap node, which answers with a collection of CAN nodes' IP addresses.

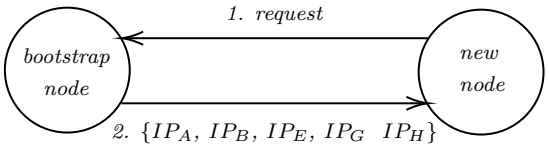

Figure 1: Bootstrap phase

Finding a Zone The new node randomly chooses a point P in the space and sends a `JOIN` request into the system via any existing node. Each node in the path then uses the CAN routing mechanism to forward the message, until it reaches the node in whose zone P lies. This current occupant node then splits its zone in half and assigns one half to the new node (the split is done assuming a certain ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave). The (key, value) pairs from the half zone to be handed over are also transferred to the new node.

Figure 2 shows the new node sending a `JOIN` request into the CAN. In the schema is supposed that the point P chosen by the entering node, lies in the zone owned by the node D; the request is sent through the node B, which was chosen between the nodes indicated by the bootstrap node. The path followed by the request to reach the node D is just a sample routing.

Figure 3 represents the space after that node D has split its zone in half and has assigned one half to the new node. It is supposed that the pairs which have to be transferred to the new node, because they belong to its zone, are $\{p_1, p_2, p_3\}$

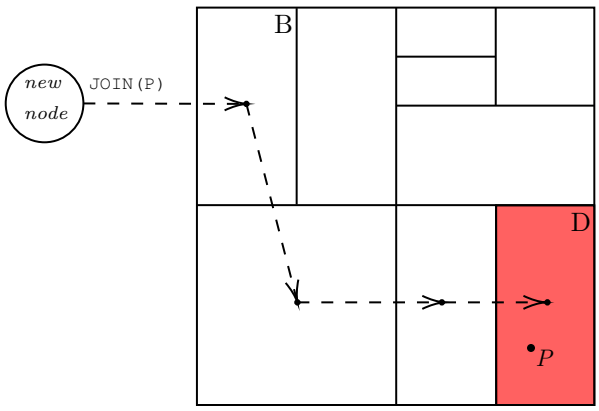

Figure 2: `JOIN` request routed to node in whose zone P lies.

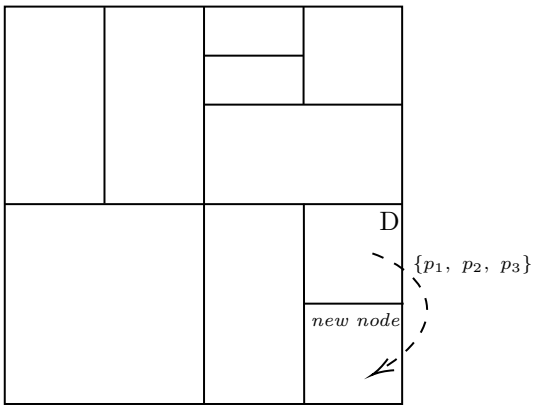

Figure 3: The zone where P was lying is split in half. One half is assigned to the new node and the (key, value) pairs relative to its zone are transferred from D.

Joining the Routing The new node learns the IP addresses of its coordinate neighbor set from the previous occupant. This set is a subset of the previous occupant's neighbors, plus that occupant itself. Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors. Finally, both the new and old nodes' neighbors must be informed of this reallocation of space. Every node in the system sends an immediate update message, followed by periodic refreshes, with its currently assigned zone to all its neighbors. These soft-state style updates ensures that all of their neighbors will quickly learn about the change and will update their own neighbor sets accordingly.

Figure 4 shows the neighbors of D and the new node. After an exchange of messages between D, F, H and the new node, the neighbor sets will become consistent. Then, the neighbors of D are F, H and the new node, whereas the neighbors of the joined node are D and H.

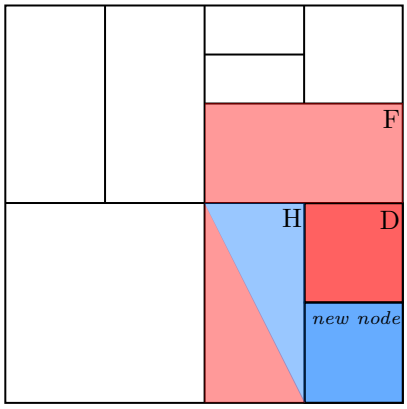

Figure 4: After a message exchange, the neighbors sets of nodes D, F, H and the new node are consistent.

The addition of a new node affects only a limited number of existing nodes in a small locality of the coordinate space. The number of neighbors a node maintains depends only on the dimensionality of the coordinate space and is independent of the total number of nodes in the system. Thus, node insertion affects only $O(d)$ existing nodes, where d is the number of dimensions of the Cartesian space.

An improvement to the joining procedure to achieve load balancing, can be applied in the splitting step: instead of directly splitting its own zone, the existing occupant node where the target point P lies, first compares the volume of its zone with those of its immediate neighbors in the coordinate space. The zone that is split to accommodate the new node is then the one with the largest volume.

3 Node departing

Node departing can happen in two different situations: a node intentionally wants to leave the CAN or a failure occurs. As in the section above, the description is exemplified in a 2-d space CAN instance.

3.1 Intentional departing

When a node intentionally wants to leave the CAN it explicitly hand over its zone and associated (key, value) pairs to one of its neighbors. If the zone of one of the neighbors can be merged with the departing node's zone to produce a valid single zone (Fig. 5), then this is done. If not, then the zone is handed to the neighbor whose current zone is smallest, and that node will then temporarily handle both zones (Fig. 6).

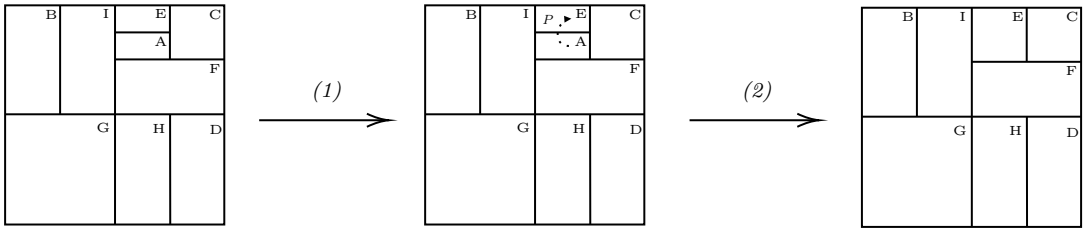

Figure 5: Node A wants to leave. Node E zone can be merged with the departing node's zone: first A transfers the (key, value) pairs P associated to its zone (1) to E, then A leaves (2).

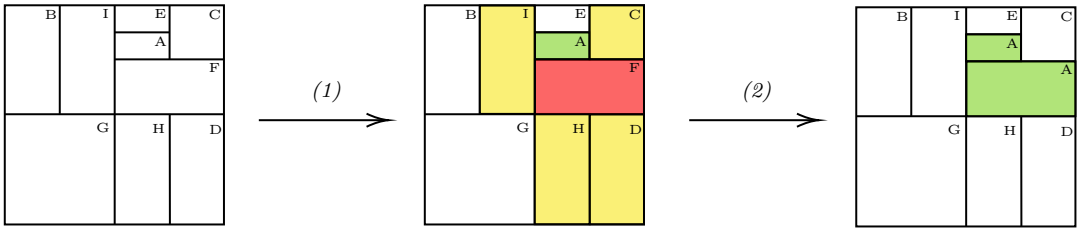

Figure 6: Node F wants to leave. Its zone can't be merged with any of the neighbors' ones. The neighbors of F are A, C, D, H, I and the one with the smallest zone is A (1). Then A is chosen and will temporarily handle its zone and the one left from F (2). The pairs held by node F are transferred to node A before its departure.

3.2 Node failure

When a node failure occurs, an immediate takeover algorithm is applied to ensure one of the failed node's neighbors takes over the zone. In this case the (key, value) pairs held by the departing node would be lost until the state is refreshed by the holders of the data. A node failure can be detected by its neighbors from the prolonged absence of update messages from it.

Once a node has decided that its neighbor has died, it initiates the takeover mechanism and starts a takeover timer running. Each neighbor of the failed node will do this independently, with the timer initialized in proportion to the volume of the node's own zone. When the timer expires, a node sends a `TAKEOVER` message conveying its own zone volume to all of the failed node's neighbor. On receipt of this message, a node cancels its own timer if the zone volume in the message is smaller that its own zone volume, or it replies with its own `TAKEOVER` message. In this way, a neighboring node is efficiently chosen which is still live, and which has a small zone volume.

The following picture shows an example where a single node fails and the takeover algorithm is used to handle the uncovered zone.

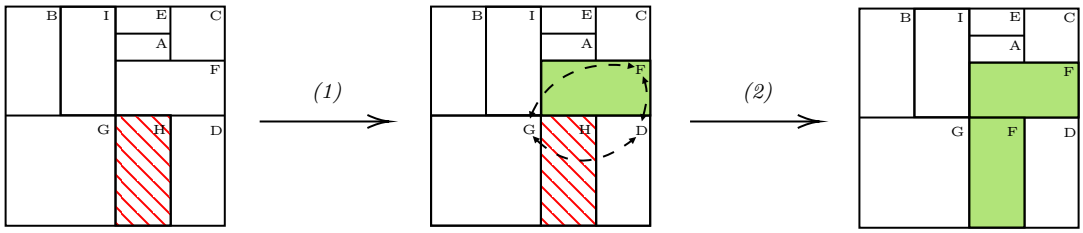

Figure 7: Node H fails. The failure is detected by the neighbors of H, which are D, F and G. They independently start the takeover algorithm and, for each one, once the timer expires they send a `TAKEOVER` message to the other neighbors of the failed node (1). Neighbors D and F have the same volume and, supposing the node F starts the takeover algorithm before D, it will be chosen to handle the uncovered zone (2).

Under certain failure scenarios involving the simultaneous failure of multiple adjacent nodes, it is possible that a node detects a failure, but that less than half of the failed node's neighbors are still reachable. If it takes over under these circumstances, it is possible for the CAN state to become inconsistent. In such cases, prior to triggering the repair mechanism, the node performs an expanding ring search for any nodes residing beyond the failure region and hence it eventually rebuilds sufficient neighbor state to initiate a takeover safely.

Both the normal leaving procedure and the immediate takeover algorithm can result in a node holding more than one zone. To prevent repeated further fragmentation of the space, a background zone-reassignment algorithm runs to ensure that the CAN tends back towards one zone per node. This algorithm aims at maintaining, even in the face of node failures, a dissection of the coordinate space that could have been created solely by nodes joining the system.

4 Questions

4.1 Question 1

Q Explain whether the hashing function used in CAN may be defined a consistent hashing.

A To be defined consistent, an hashing function must guarantee that adding / removing nodes implies moving only a minority of data items. In particular, the following properties must be satisfied:

1. Distribution scheme does not depend directly on the number of servers.
2. Each node manages an interval of consecutive hash keys, not a set of sparse keys.
3. Intervals are joined / split when nodes join / leave the network and keys redistributed between adjacent peers.

The hashing function used in CAN maps a (key, value) pair to a given point in the d-dimensional coordinate space. The space is divided in zones, each one controlled by a node. A given pair will always be mapped to the same point in space and will be independent to the number of nodes currently in the system, then property (1) is satisfied.

In CAN, to each node is assigned a contiguous portion of the coordinate space, then it will manage all the consecutive pairs lying on his zone and not a set of sparse (key, value) pairs; property (2) is satisfied.

When a node joins the CAN, the node who owns the zone chosen by the joining node splits its zone, assigns half of it to the new node and hand over the relative (key, value) pairs. When a node intentionally leaves the system, its zone is merged, if possible, to one of the neighbors' zone; if it is not possible the zone is temporarily assigned to a neighbor. In both cases the (key, value) pairs are transferred to the neighbor before departing. Instead, if a node fails, an immediate takeover algorithm is executed in order to assign the uncovered zone to a neighbor of the failed node. In this case the (key, value) pairs held by the departing node would be lost until the state is refreshed by the holders of the data. Also the property (3) is satisfied by the hashing function used in CAN, then it can be defined a consistent hashing.

4.2 Question 2

Q Compare the routing algorithm of CAN and Kademlia, in particular compare their complexity.

A Basic routing in CAN works by following the straight line path through the Cartesian space from source to destination coordinates. A CAN message includes the destination coordinates; using its neighbor coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. For a d-dimensional space partitioned into n equal zones the routing path length is thus $O(d(n^{1/d}))$.

Instead, Kademlia is a prefix matching DHT, which means that the nodes and keys are mapped to numbers of m digits in base b, and each key is assigned to the node with which shares the longest prefix, if possible, or the node with numerically close ID. The identifier space is modelled by a tree of depth $\leq m$, where each node has b sons. The routing algorithm used in Kademlia is Plaxton routing: given a key, at each step at least b bit of the prefix are corrected; at each correction, routing moves to the node whose has in common with the key the considered prefix. The number of look-up hops, and then the routing path length is $O(log_b(n))$, where n is the number of nodes in the system.

Comparing the complexity of routing in CAN and Kademlia we have that:

- Routing in CAN requires $O(d(n^{1/d}))$ steps, where d is the number of dimensions and n the number of nodes.
- Routing in Kademlia requires $O(log_b(n))$ steps, where b is the base of the hashes and n the number of nodes.

If we fix the number of nodes n, we can compare the complexity of the two routing strategies: if the number of dimensions d in CAN is fixed to $d = log_b(n)/2$, then the number of steps coincides to $O(log_b(n))$, obtaining similar performance in CAN and Kademlia.

4.3 Question 3

Q Suppose that a CAN node is behind a NAT: describe at least one technique to transverse the NAT.

A There are three main mechanisms to transverse a NAT: relaying, connection reversal, hole punching.

The relaying technique uses an intermediate relay node; if two nodes wants to exchange information between each other and at least one is natted, they can both open a connection to the relay node (identified by a public IP address) and all the traffic between the two peers goes through the relay.

The connection reversal mechanism still uses a relay node and can be used only between a public and a natted peer. The peer with private address previously opens a connection to the relay node. When the peer with public IP wants to open a connection to the natted peer, it asks the relay to tell the node with private address to open an outgoing connection toward itself, sending its address to the relay. Then the relay asks for a connection reversal to the natted peer and once the connection is established, the communication will happen directly between the two peers.

The hole punching technique allows a direct communication between two peers by exploiting a relay server. Also in this case, the relay server is used only to put the peers in contact, but packets are directly

sent between the two peers. This mechanism works in several scenarios, included the case of one peer behind the NAT and the other public, which is the one of interest in case of a CAN node behind a NAT. This last technique differs by the kind of NAT used:

- In full cone NAT, the natted host sends a message to the relay server, which will store its NAT Endpoint (public IP address and port); when the public host wants to transfer data with the natted one, it asks its NAT Endpoint to the relay server and the communication between the two hosts will happen through the same NAT Endpoint.
- In restricted cone NAT, the natted host has previously communicated with the rendezvous server; when the public host wants to exchange data with the natted host, it will send a request to the relay server, which will forward it to the host with private IP. At this point the natted host will send a packet to the public host, opening a "hole" in the NAT that can be used by the public node to communicate.
- If the NAT which the CAN node is behind is symmetric, the hole punching technique can not be used: in this case the NAT assigns new mapping for different destination and only the recipient of the message can send a packet back to the internal host. The rendezvous mechanism is then not useful and in this case connection reversal or relaying should be used.