notes

alex

December 13, 2018

Contents

1 Structured P2P Networks

TODO, potentially read all of the experiments performed in pastry. Potentially not, who cares. Also the math in Kademlia.

1.1 Chord

1.1.1 Introduction

- A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item.
- Chord provides support for just one operation: given a key, it maps the key onto a node.

- Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps.
- Peer-to-peer systems and applications are distributed systems without any centralised control or hierarchical structure or organisation and each peer is equivalent in functionality.
- Peer-to-peer applications can promote a lot of features, such as redundant storage, permanence, selection of nearby server, anonymity, search, authentication and hierarchical naming (note the structure is still peer-to-peer, the names are just attributes and data the peers hold)
- Core operation in most P2P systems is efficient location of data items.
- Chord is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.
- Chord uses a variant of consistent hashing to assign keys to Chord nodes.
- Consistent hashing is a special kind of hashing such that when a hash table is resized, only K/n keys need to be remapped on average, where K is the number of keys and n is the number of slots.
- Additionally, consistent hashing tends to balance load, as each node will receive roughly the same amount of keys.
- Each Chord node needs "routing" information about only a few others nodes, leading to better scaling.
- Each node maintains information about O(log N) other nodes and resolves lookups via O(log N) messages. A change in the network results in no more than $O(\log^2 N)$ messages.
- Chords performance degrades gracefully, when information is out of date in the nodes routing tables. It's difficult to maintain consistency of O(log N) states. Chord only requires one piece of information per node to be correct, in order to guarantee correct routing.
- Finger tables only forward looking
- I.e messages arriving at a peer tell it nothing useful, knowledge must be gained explicitly

- Rigid routing structure
- Locality difficult to establish

1.1.2 System Model

- Chord simplifies the design of P2P systems and applications based on it by addressing the following problems:

    1. **Load balance:** Chord acts as a Distributed Hash Function, spreading keys evenly over the nodes, which provides a natural load balance
    2. **Decentralization:** Chord is fully distributed. Improves robustness and is nicely suited for loosely-organised p2p applications
    3. **Scalability:** The cost of a lookup grows as the log of the number of nodes
    4. **Availability:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensur- ing that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
    5. **Flexible naming:** No constraints on the structure of the keys it looks up

1. Use cases of Chord

    - Cooperative Mirroring: Essentially a load balancer
    - Time-Shared Storage: If a person wishes some data to be always available machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down.
    - Distributed Indexes: A key in this application could be a few keywords and values would be machines offering documents with those keywords
    - Large-scale Combinatorial Search: Keys are candidate solutions to the problem; Chord maps these keys to the machines responsible for testing them as solutions.

1.1.3 The Base Chord Protocol

- The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes.

1. Overview

    - Chord improves the scalability of consistent hashing by avoid- ing the requirement that every node know about every other node.

2. Consistent Hashing

    - The consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key.
    - Identifiers are ordered in an identifier circle modulo 2^m
    - Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space.
    - This node is called the successor node of key k, succ(k). It's the first node clockwise from k, if identifiers are presented as a circle.
    - To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor.
    - The claims about the effeciency of consistent hashing, relies on the identifiers being chosen uniformly randomly. SHA-1 is very much deterministic, as is all hash functions. As such, an adversary could in theory pick a bunch of identifiers close to each other and thus force a single node to carry a lot of files, ruining the balance. However, it's considered difficult to break these hash functions, as such we can't produce files with specific hashes.
    - When consistent hashing is implemented as described above, the theorem proves a bound of eps = O(log N). The consistent hashing paper shows that eps can be reduced to an arbitrarily small constant by having each node run O(log N) "virtual nodes" each with its own identifier.
    - This is difficult to pre-determine, as the load on the system is unknown a priori.

3. Scalable Key Location

   - A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle.
   - Queries for a given identifier can be passed around the circle via these suc- cessor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to.
   - To avoid having to potentialy traverse all N nodes, if the identifiers are "unlucky", Chord maintains extra information.
   - m is the number of bits in the keys
   - Each node n maintains a routing table with at most m entries, called the finger table.
   - The i'th entry in the table at node n contains the identity of the first node, s, that succeeds n by at least $2^{(i-1)}$ on the identifier circle, s = succ(n+$2^{(i-1)}$) for 1 <= i <= m and everything i mod 2^m
   - Node s in the ith finger of node n is n.finger[i].node
   - A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.
   - First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away.
   - The nodes keep an interval for each key implicitly, which essentially covers the keys that the the specific key is the predecessor for. This allows for quickly looking up a key, if it's not known, since one can find the interval which contains it!
   - The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in find predecessor to halve the distance to the target identifier.

4. Node Joins

   - In dynamic networks, nodes can join and leave at any time. Thus the main challenge is to preserve the ability to lookup of every key.
   - There are to invariants:

     (a) Each node's succ is correctly maintained

(b) For every key k, node succ(k) is responsible for k

- We also want the finger tables to be correct
- To simplify the join and leave mechanisms, each node in Chord maintains a predecessor pointer.
- To preserve the invariants stated above, Chord must perform three tasks when a node n joins the network:
  (a) Initialise the predecessor and fingers of node n
  (b) Update the fingers and predecessors of existings node to reflect the addition of n
  (c) Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

(a) Initializing fingers and predecessor

- Node n learns its pre- decessor and fingers by asking n' to look them up

(b) Updating fingers of existing nodes

- Thus, for a given n, the algorithm starts with the ith finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose ith finger precedes n.

(c) Transfering keys

- Node n contacts it's the node immediately following itself and simply asks for the transfering of all appropriate values

1.1.4 Concurrenct operations and failures

1. Stabilitzation

- The join algorithm in Section 4 aggressively maintains the finger tables of all nodes as the network evolves. Since this invariant is difficult to maintain in the face of concurrent joins in a large network, we separate our correctness and performance goals.
- A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct fin- ger table entries, which allows these lookups to be fast as well as correct.

- Joining nodes can affect performance in three ways, all tables are still correct and result is found, succ is correct but fingers aren't, result will still be found and everything is wrong, in which case nothing might be found. The lookup can then be retried shortly after.
- Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes
- We have not discussed the adjustment of fingers when nodes join because it turns out that joins don't substantially damage the per- formance of fingers. If a node has a finger into each interval, then these fingers can still be used even after joins.

2. Failures and Replication

   - When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing.
   - The key step in failure recovery is maintaining correct successor pointers
   - To help achieve this, each Chord node maintains a "successor-list" of its r nearest successors on the Chord ring.
   - If node n notices that its successor has failed, it replaces it with the first live en- try in its successor list. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, stabilize will correct finger table entries and successor-list entries pointing to the failed node.

1.1.5 Simulations and Experimental Results

- The probability that a particular bin does not contain any is for large values of N approximately 0.368
- As we discussed earlier, the consistent hashing paper solves this prob- lem by associating keys with virtual nodes, and mapping mul- tiple virtual nodes (with unrelated identifiers) to each real node. Intuitively, this will provide a more uniform coverage of the iden- tifier space.

1.1.6 Conclusion

- Attractive features of Chord include its simplicity, provable cor- rectness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, al- beit at degraded performance, when a node's information is only partially correct. Our theoretical analysis, simulations, and exper- imental results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recov- ery.

1.2 Pastry

- Pastry, a scalable, distributed object location and routing substrate for wide-area peer-to-peer applications.
- It can be used to support a variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming.
- Each node in the Pastry network has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries.
- Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a to scalar proximity metric like the number of IP routing hops.
- Experimental results obtained with a prototype implementation on an emulated network of up to 100,000 nodes confirm Pastry's scalability and efficiency, its ability to self-organize and adapt to node failures, and its good network locality properties.

1.2.1 Introduction

- Pastry is completely decentralized, fault-resilient, scalable, and reliable. Moreover, Pastry has good route locality properties.

- Pastry is intended as general substrate for the construction of a variety of peer-to- peer Internet applications like global file sharing, file storage, group communication and naming systems.
- Several application have been built on top of Pastry to date, including a global, persistent storage utility called PAST [11, 21] and a scalable publish/subscribe system called SCRIBE [1]. Other applications are under development.
- Each node in the Pastry network has a unique numeric identifier (nodeId)
- When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes.
- The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network.
- At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message.
- Pastry takes into account network locality; it seeks to minimize the distance mes- sages travel, according to a scalar proximity metric like the number of IP routing hops.
- Because nodeIds are randomly assigned, with high probability, the set of nodes with adjacent nodeId is diverse in geography, ownership, jurisdiction, etc. Applications can leverage this, as Pastry can route to one of nodes that are numerically closest to the key.
- A heuristic ensures that among a set of nodes with the closest nodeIds to the key, the message is likely to first reach a node "near" the node from which the message originates, in terms of the proximity metric.

1. PAST
    - PAST, for instance, uses a fileId, computed as the hash of the file's name and owner, as a Pastry key for a file. Replicas of the file are stored on the k Pastry nodes with nodeIds numerically closest to the fileId. A file can be looked up by sending a message

[1] DEFINITION NOT FOUND.

via Pastry, using the fileId as the key. By definition, the lookup is guaranteed to reach a node that stores the file as long as one of the k nodes is live.

- Moreover, it follows that the message is likely to first reach a node near the client, among the k nodes; that node delivers the file and consumes the message. Pastry's notification mechanisms allow PAST to maintain replicas of a file on the nodes closest to the key, despite node failure and node arrivals, and using only local coordination among nodes with adjacent nodeIds.

2. SCRIBE

   - in the SCRIBE publish/subscribe System, a list of subscribers is stored on the node with nodeId numerically closest to the topicId of a topic, where the topicId is a hash of the topic name. That node forms a rendez-vous point for publishers and subscribers. Subscribers send a message via Pastry using the topicId as the key; the registration is recorded at each node along the path. A publisher sends data to the rendez-vous point via Pastry, again using the topicId as the key. The rendez-vous point forwards the data along the multicast tree formed by the reverse paths from the rendez-vous point to all subscribers.

1.2.2 Design of Pastry

- A Pastry system is a self-organizing overlay network of nodes, where each node routes client requests and interacts with local instances of one or more applications.

- Each node in the Pastry peer-to-peer overlay network is assigned a 128-bit node identifier (nodeId).

- The nodeId is used to indicate a node's position in a circular nodeId space, which ranges from 0 to 2^{128} - 1 (sounds like a modular ring type thing, as in Chord).

- Nodeids are distributed uniformly in the 128-bit nodeid space, such as computing the hash of IP.

- As a result of this random assignment of nodeIds, with high probability, nodes with adjacent nodeIds are diverse in geography, ownership, jurisdiction, network attachment, etc.

- Under normal conditions, in a network of N nodes, Pastry can route to the numerically closest node to a given key in less than $\log_{(2^b)}$ N steps. b is some random configuration parameter.
- For the purpose of routing, nodeIds and keys are thought of as a sequence of digits with base 2^b.
- In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or bits) longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. To support this routing procedure, each node maintains some routing state
- Despite concurrent node failures, eventual delivery is guaranteed unless |L|/2 nodes with adjacent nodeIds fail simul- taneously (|L| is a configuration parameter with a typical value of 16 or 32).

1. Pastry Node State
    - Each Pastry node maintains a routing table, a neighborhood set and a leaf set.
    - A node's routing table, R, is organized into $\log_{(2^b)}$ N rows with 2^b - 1 entries each.
    - The 2^b - 1 entries at row n each refer to a node whose nodeid shares the present node's nodeid in the first n digits, but whose n+1th digit has one of the 2^b - 1 possible values other than then n+1th digit in the present node's id.
    - Each entry in the routing table contains the IP address of one of potentially many nodes whose nodeId have the appropriate prefix; in practice, a node is chosen that is close to the present node, according to the proximity metric.
    - If no node is known with a suitable nodeId, then the routing table entry is left empty.
    - The neighborhood set M contains the nodeIds and IP addresses of the |M| nodes that are closest (according the proximity metric) to the local node.
    - Applications are responsible for providing proximity metrics

- The neighborhood set is not normally used in routing messages; it is useful in maintaining locality properties
- The leaf set L is the set of nodes with the |L|/2 numerically closest larger nodeIds, and the |L|/2 nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. The leaf set is used during the message routing

2. Routing

   - Given a message, the node first checks to see if the key falls within the range of nodeIds covered by its leaf set
   - If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key (possibly the present node)
   - If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit
   - In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the local node, and is numerically closer to the key than the present node's id.
   - Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless |L|/2 adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be live.
   - It can be shown that the expected number of routing steps is $\log_{(2^b)}$ N steps.
   - If a message is forwarded using the routing table, then the set of nodes whose ids have a longer prefix match with the key is reduced by a factor of 2^b in each step, which means the destination is reached in $\log_{(2^b)}$ N steps.
   - If the key is within range of the leaf set, then the destination node is at most one hop away.
   - The third case arises when the key is not covered by the leaf set (i.e., it is still more

   than one hop away from the destination), but there is no routing table entry. Assuming accurate routing tables and no recent node failures, this means that a node with the appropriate prefix does not exist.

- The likelihood of this case, given the uniform distribution of nodeIds, depends on |L|. Analysis shows that with $|L| = 2^b$ and $|L| = 2 * 2^b$, the probability that this case arises during a given message transmission is less than .02 and 0.006, respectively. When it happens, no more than one additional routing step results with high probability.

3. Pastry API

   - Substrate: not an application itself, rather it provides Application Program Interface (API) to be used by applications. Runs on all nodes joined in a Pastry network
   - Pastry exports following operations; nodeId and route.
   - Applications layered on top of PAstry must export the following operations; deliver, forward, newLeafs.

4. Self-organization and adaptation

   (a) Node Arrival

   - When a new node arrives, it needs to initialize its state tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby Pastry node A, according to the proximity metric, that is already part of the system.
   - Let us assume the new node's nodeId is X.
   - Node X then asks A to route a special "join" message with the key equal to X. Like any message, Pastry routes the join message to the existing node Z whose id is numerically closest to X.
   - In response to receiving the "join" request, nodes A, Z, and all nodes encountered on the path from A to Z send their state tables to X.
   - Node X initialized its routing table by obtaining the i-th row of its routing table from the i-th node encountered along the route from A to Z to
   - X can use Z's leaf set as basis, since Z is closest to X.
   - X use A's neighborhood to initialise its own
   - Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing

table. Those nodes in turn update their own state based on the information received.

(b) Node Depature

- A Pastry node is considered failed when its immediate neighbors in the nodeId space can no longer communicate with the node.
- To replace a failed node in the leaf set, its neighbor in the nodeId space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf table.
- The failure of a node that appears in the routing table of another node is detected when that node attempts to contact the failed node and there is no response.
- To replace a failed node in a routing table entry, a node contacts the other nodes in the row of the failed node and asks if any of them knows a node with the same prefix.
- a node attempts to contact each member of the neighborhood set periodically to see if it is still alive.

5. Locality

- Pastry's notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance.
- It is assumed that the application provides a function that allows each Pastry node to determine the "distance" of a node with a given IP address to itself.
- Throughout this discussion, we assume that the proximity space defined by the cho- sen proximity metric is Euclidean; that is, the triangulation inequality holds for dis- tances among Pastry nodes.
- If the triangulation inequality does not hold, Pastry's basic routing is not affected; however, the locality properties of Pastry routes may suffer.

(a) Route Locality

- although it cannot be guaranteed that the distance of a message from its source increases monotonically at each step, a

message tends to make larger and larger strides with no possibility of returning to a node within d_i of any node i encountered on the route, where d_i is the distance of the routing step taken away from node i. Therefore, the messag ehas nowhere to go but towards its destination.

(b) Locating the nearest among k nodes

- Recall that Pastry routes messages towards the node with the nodeId closest to the key, while attempting to travel the smallest possible distance in each step.
- Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no sense of global direction.

6. Arbitrary node failures and network partitions

- As routing is deterministic by default, a malicious node can fuck things up. Randomized routing fixes this.
- Another challenge are IP routing anomalies in the Internet that cause IP hosts to be unreachable from certain IP hosts but not others.
- However, Pastry's self-organization protocol may cause the creation of multiple, isolated Pastry overlay networks during periods of IP routing failures. Because Pastry relies almost exclusively on information exchange within the overlay network to self-organize, such isolated overlays may persist after full IP connectivity resumes.
- One solution to this problem involves the use of IP multicast.

1.2.3 Conclusion

- This paper presents and evaluates Pastry, a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably routes a message to the live node with a nodeId numerically closest to a key. Pastry can be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems. Results with as many as 100,000 nodes in an emulated network confirm that Pastry is efficient and scales well,

that it is self-organizing and can gracefully adapt to node failures, and that it has good locality properties.

1.3 Kademlia

1.3.1 Abstract

- A peer-to-peer distributed hash table with provable consistency and performance in a fault-prone environment
- system routes queries and locates nodes using a novel XOR-based metric topology
- The topology has the property that every message exchanged conveys or reinforces useful contact information.
- The system exploits this information to send parallel, asynchronous query messages that tolerate node failures without imposing timeout delays on users.

1.3.2 Introduction

- Kademlia is a P2P DHT
- Kademlia has a number of desirable features not simultaneously offered by any previous DHT. It minimizes the number of configuration messages nodes must send to learn about each other.
- Configuration information spreads automatically as a side-effect of key lookups.
- Kademlia uses parallel, asynchronous queries to avoid timeout delays from failed nodes.
- Keys are opaque, 160-bit quantities (e.g., the SHA-1 hash of some larger data)
- Participating computers each have a node ID in the 160-bit key space.
- (key,value) pairs are stored on nodes with IDs "close" to the key for some notion of closeness.
- XOR is symmetric, allowing Kademlia participants to receive lookup queries from precisely the same distribution of nodes contained in their routing tables

- Without this property, systems such as Chord do not learn useful routing information from queries they receive.
- Worse yet, asymmetry leads to rigid routing tables. Each entry in a Chord node's finger table must store the precise node preceding some interval in the ID space.
- Each entry in a Chord node's finger table must store the precise node preceding some interval in the ID space. Any node actually in the interval would be too far from nodes preceding it in the same interval. Kademlia, in contrast, can send a query to any node within an interval, allowing it to select routes based on latency or even send parallel, asynchronous queries to several equally appropriate nodes.
- Kademlia most resembles Pastry's first phase, which (though not described this way by the authors) successively finds nodes roughly half as far from the target ID by Kademlia's XOR metric.
- In a second phase, however, Pastry switches distance metrics to the numeric difference between IDs. It also uses the second, numeric difference metric in replication. Unfortunately, nodes close by the second metric can be quite far by the first, creating discontinuities at particular node ID values, reducing performance, and complicating attempts at formal analysis of worst-case behavior.

1.3.3 System Description

- Kademlia assign 160-bit opaque IDs to nodes and provide a lookup algorithm that locates successively "closer" nodes to any desired ID, converging to the lookup target in logarithmically many steps
- An identifier is opaque if it provides no information about the thing it identifies other than being a seemingly random string or number
- Kademlia effectively treats nodes as leaves in a binary tree, with each node's position determined by the shortest unique prefix of its ID
- For any given node, we divide the binary tree into a series of successively lower subtrees that don't contain the node. The highest subtree consists of the half of the binary tree not containing the node.
- The next subtree consists of the half of the remaining tree not containing the node, and so forth

- The Kademlia protocol ensures that every node knows of at least one node in each of its subtrees, if that subtree contains a node. With this guarantee, any node can locate any other node by its ID

1. XOR Metric

    - Each Kademlia node has a 160-bit node ID. Node IDs are currently just random 160-bit identifiers, though they could equally well be constructed as in Chord.
    - Every message a node transmits includes its node ID, permitting the recipient to record the sender's existence if necessary.
    - Keys, too, are 160-bit identifiers. To assign hkey,valuei pairs to particular nodes, Kademlia relies on a notion of distance between two identifiers. Given two 160-bit identifiers, x and y, Kademlia defines the distance between them as their bitwise exclusive or (XOR) intepreted as an integer.
    - XOR is nice, as it is symmetric, offers the triangle property even though it's non-euclidean.
    - We next note that XOR captures the notion of distance implicit in our binary-tree-based sketch of the system.
    - In a fully-populated binary tree of 160-bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing them both. When a tree is not fully populated, the closest leaf to an ID x is the leaf whose ID shares the longest common prefix of x.
    - Overlap in regards to closest might happen. In this case the closest leaf to x will be the closest leaf to ID x~ produced by flipping the bits in corresponding to the empty branches of the tree (???)
    - Like Chord's clockwise circle metric, XOR is unidirectional. For any given point x and distance > 0, there is exactly one point y such that d(x, y) = . Unidirectionality ensures that all lookups for the same key converge along the same path, regardless of the originating node. Thus, caching hkey,valuei pairs along the lookup path alleviates hot spots.

2. Node state

- For each 0 i < 160, every node keeps a list of (IP address, UDP port, Node ID) triples for nodes of distance between 2^i and 2^i+1 from itself. We call these lists k-buckets.
- Each k-bucket is kept sorted by time last seen—least-recently seen node at the head, most-recently seen at the tail. For small values of i, the k-buckets will generally be empty (as no appropriate nodes will exist). For large values of i, the lists can grow up to size k, where k is a system-wide replication parameter.
- k is chosen such that it is unlikely that k nodes will fail at the same time.
- When a message is received, request or reply, from another node, the receiver updates its appropriate k-bucket, for the sender's node id. If the node is already present there, it's moved to the tail, if it's not there and there is room, it's inserted. If the bucket is full, the least recently seen node is pinged, if it doesn't respond, it gets replaced, if it does respond, the new node is discarded.
- k-buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the list.
- This works well for systems with an otherwise high churn rate, as nodes who are alive for a longer period, are more likely to stay alive.
- A second benefit of k-buckets is that they provide resistance to certain DoS attacks. One cannot flush nodes' routing state by flooding the system with new nodes, as new nodes are only inserted, once the old ones die.

3. Kademlia Protocol

- The Kademlia protocol consists of four RPCs: ping, store, find node, and find value.
- The ping RPC probes a node to see if it is online.
- store a node to store a (key, value) pair for later retrieval
- find node takes a 160-bit ID as an argument. The recipient of the RPC returns (IP address, UDP port, Node ID) triples for the k nodes it knows aboutclosest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes

in all its k-buckets combined, in which case it returns every node it knows about).

- find value behaves like find node—returning (IP address, UDP port, Node ID) triples—with one exception. If the RPC recipient has received a store RPC for the key, it just returns the stored value.
- In all RPCs, the recipient must echo a 160-bit random RPC ID, which provides some resistance to address forgery. pings can also be piggy-backed on RPC replies for the RPC recipient to obtain additional assurance of the sender's network address.

(a) Node lookup

i. Node lookup is performed recursively. The lookup initiator starts by picking alpha nodes from its closest k-bucket (is the closest to the iniator or closest to the node we wish to lookup ??).

ii. The iniator then sends parallel async $\text{find}_{\text{node}}$ RPCs to these alpha nodes.

iii. In the recursive step, the initiator resends the find node to nodes it has learned about from previous RPCs. (This recursion can begin before all of the previous RPCs have returned).

iv. If a response is not found the alpha nodes queried, the iniator instead query all of the k nodes which were returned.

v. Lookup terminates when all k has responded or failed to respond.

vi. When $= 1$, the lookup algorithm resembles Chord's in terms of message cost and the latency of detecting failed nodes. However, can route for lower latency because it has the flexibility of choosing any one of k nodes to forward a request to.

(b) Store

- Most operations are implemented in terms of the above lookup procedure. To store a (key,value) pair, a participant locates the k closest nodes to the key and sends them store RPCs
- Additionally, each node re-publishes (key,value) pairs as necessary to keep them alive

- For file sharing, it's required that the original publisher of a (key,value) pair to republish it every 24 hours. Otherwise, (key,value) pairs expire 24 hours after publication, so as to limit stale index information in the system.

(c) Find value

- To find a (key,value) pair, a node starts by performing a lookup to find the k nodes with IDs closest to the key. However, value lookups use find value rather than find node RPCs. Moreover, the procedure halts immediately when any node returns the value. For caching purposes, once a lookup succeeds, the requesting node stores the (key,value) pair at the closest node it observed to the key that did not return the value.
- Because of the unidirectionality of the topology, future searches for the key are likely to hit cached entries before querying the closest node.
- To avoid overcaching, the expiration time of any key-value pair is determined by the distance between the current node and the node whose ID is closest to the key ID.

(d) Refreshing buckets

- To handle pathological cases in which there are no lookups for a particular ID range, each node refreshes any bucket to which it has not performed a node lookup in the past hour. Refreshing means picking a random ID in the bucket's range and performing a node search for that ID

(e) Joining network

- To join the network, a node u must have a contact to an already participating node w. u inserts w into the appropriate k-bucket. u then performs a node lookup for its own node ID. Finally, u refreshes all k-buckets further away than its closest neighbor. During the refreshes, u both populates its own k-buckets and inserts itself into other nodes' k-buckets as necessary.

4. Routing Table

- The routing table is a binary tree whose leaves are k-buckets.
- each k-bucket covers some range of the ID space, and together the k-buckets cover the entire 160-bit ID space with no overlap.

- When a node u learns of a new contact and this can be inserted into a bucket, this is done. Otherwise, if the k-bucket's range includes u's own node ID, then the bucket is split into two new buckets, the old contents divided between the two, and the insertion attempt repeated. This is what leads to one side of the binary tree being one large bucket, as it won't get split
- If tree is highly unbalanced, issues may arise (what issues ??). To avoid these, buckets may split, regardless of the node's own ID residing in these.
- nodes split k-buckets as required to ensure they have complete knowledge of a surrounding subtree with at least k nodes.

5. Efficient key re-publishing

   - Keys must be periodically republished as to avoid data disappearing from the network or that data is stuck on un-optimal nodes, as new nodes closer to the data might join the network.
   - To compensate for nodes leaving the network, Kademlia republishes each key-value pair once an hour.
   - As long as republication intervals are not exactly synchronized, only one node will republish a given key-value pair every hour.

1.3.4 Implementation Notes

1. Optimized contact accounting

   - To reduce traffic, Kademlia delays probing contacts until it has useful messages to send them. When a Kademlia node receives an RPC from an unknown contact and the k-bucket for that contact is already full with k entries, the node places the new contact in a replacement cache of nodes eligible to replace stale k-bucket entries.
   - When a contact fails to respond to 5 RPCs in a row, it is considered stale. If a k-bucket is not full or its replacement cache is empty, Kademlia merely flags stale contacts rather than remove them. This ensures, among other things, that if a node's own network connection goes down teporarily, the node won't completely void all of its k-buckets.
   - This is nice because Kademlia uses UDP.

2. Accelerated lookups

    - Another optimization in the implementation is to achieve fewer hops per lookup by increasing the routing table size. Conceptually, this is done by considering IDs b bits at a time instead of just one bit at a time
    - This also changes the way buckets are split.
    - This also changes the XOR-based routing apparently.

1.3.5 Summary

- With its novel XOR-based metric topology, Kademlia is the first peer-to-peer system to combine provable consistency and performance, latency-minimizingrouting, and a symmetric, unidirectional topology. Kademlia furthermore introduces a concurrency parameter, , that lets people trade a constant factor in bandwidth for asynchronous lowest-latency hop selection and delay-free fault recovery. Finally, Kademlia is the first peer-to-peer system to exploit the fact that node failures are inversely related to uptime.

1.4 Bouvin notes

- While first generation of structured P2P networks were largely application specific and had few guarantees, usually using worst case O(N) time, the second generation is based on structured network overlays. They are typically capable of guaranteeing O(log N) time and space and exact matches.

- Much more scalable than unstructured P2P networks measured in number of hops for routing However, churn results in control traffic; slow peers can slowdown entire system (especially in Chord); weak peers may be overwhelmed by control traffic

- The load is evenly distributed across the network, based on the uniformness of the ID space More powerful peers can choose to host several virtual peers

- Most systems have various provisions for maintaining proper routing and defending against malicious peers

- A backhoe is unlikely to take out a major part of the system – at least if we store at k closest nodes

2 Mobile Ad-hoc Networks and Wireless Sensor Networks

2.1 Routing in Mobile Ad-hoc Networks

2.1.1 Introduction

- Routing is the process of passing some data, a message, along in a network.
- The message originates from a source host, travels through intermediary hosts and ends up at a destination host.
- Intermediary hosts are called routers
- Usually a few questions has to be answered when a message is routed:
  1. How do the hosts acting as routers know which way to send the message?
  2. What should be done if multiple paths connect the sender and receiver?
  3. Does an answer to the message have to follow the same path as the original message?
- Simple solution: Broadcast message, i.e. send it to every single person you know, every time.
- Creates a lot of traffic, it's also known as flooding.
- The responsibility of a routing protocol is to answer the three questions posed.

2.1.2 Basic Routing Protocols

- A routing protocol must enable a path or route to be found, through the network
- A network is usually modelled as a graph, inside the computers.
- This allows for edges to be weighted. Can either be distance, traffic or some metric like that
- There are two classes, Link State (LS) and Distance Vector (DS). Main difference being whether or not they use global information.

- Algorithms using global information are "Link State", as all nodes need to maintain state information about all links in the network.
- Distance Vector algorithms do not rely on global information.

1. Link State
    - All nodes and links with weights are known to all nodes.
    - This makes the problem a SSSP problem (single-source shortest path)

    (a) Djikstra
        - A set W is initialised, containing only the source node v.
        - In each iteration, the edge e with the lowest cost connecting W with a node u which isn't in W, is chosen and u is added to the set.
        - Algorithm loops n-1 times, and then the shortest path to all other nodes have been found.
        - Requires each router to have complete knowledge of the network.
        - Can be accomplished by broadcasting the identities and costs of all of the outgoing links to all other routers in the network. This has to be done, every time a weight or link changes.
        - Unrealistic for anything but very small networks.
        - Works great for small stable networks however.

2. Distance Vector
    - No global knowledge is needed
    - The shortest distance to any given node is calculated in cooperation between the nodes.
    - Based on Bellman-Ford.
    - Apparently original Bellman-Ford requires global knowledge. This is a knock-off algorithm.

    (a) Bellman-Ford
        - Decentralised, no global information is needed
        - Requires the information of the neighbours and the link costs between them and the node

- Each node stores a distance table
- The distance table is just a mapping between a node name and the distance to it. It's over all known nodes, so not just neighbours.
- When a new node is encountered, this is simply added
- Node sends updates to its neighbours. This message states that the distance from the node v to node u has changed. As such, the neighbours can compute their distance to this node u as well and update their table.
- This update may cause a chain of updates, as the neighbours might discover that this new distance is better than what they currently had.
- The route calculation is bootstrapped by having all nodes broadcast their distances to their neighbours, when the network is created.
- Algorithm
  i. Distance table is initialised for node x
  ii. Node x sends initial updates to neighbours
  iii. The algorithm loops, waiting for updates or link cost changes of directly connected links (neighbours (?))
  iv. Whenever either event is received, the appropriate actions are taken, such as sending updates or changing values in the distance table
- Generates less traffic, since only neighbours are needed to be known.
- Doesn't need global knowledge, general advantage in large networks or networks with high churn rate
- Doesn't have to recompute the entire distance table whenever a single value changes, as Djikstras algorithm has to.
- Suffers from the "Count-to-infinity" problem, which happens when a route pass twice through the same node and a link starts going towards infinity. If there is a network A - B - C - D. A dies. B sets the distance to infinity. When tables are shared, B sees that C knows a route to A of distance 2, as such it updates its distance to 3. (1 to C, 2 from C to A). C then has to update its distance to A to 4 and so it goes.
- A way of avoiding this, is only sending information to the neighbours that are not exclusive links to the destination, so

C shouldn't send any information to B about A, as B is the only way to A.

2.1.3 MANET Routing

- Both Djikstra and Bellman-Ford were designed to operate in fairly stable networks.
- MANETs are usually quite unstable, as possibly all nodes are mobile and may be moving during communication.
- MANETs typically consist of resource-poor, energy constrained devices with limited bandwidth and high error rates.
- Also has missing infrastructure and high mobility
- According to Belding-Royer (who he??), the focus should be on the following properties
  1. Minimal control overhead (due to limited energy and bandwidth)
  2. Minimal processing overhead (Typically small processors)
  3. Multihop routing capability (No infrastructure, nodes must act as routers)
  4. Dynamic topology maintenance (High churn rates forces topology to be dynamic and be capable of easily adapting)
  5. Loop prevention (Loops just take a lot of bandwidth)
- MANET routing protocols are typically either pro-active or re-active.

1. Proactive
   - Every node maintains a routing table of routes to all other nodes in the network
   - Routing tables are updated whenever a change occurs in the network
   - When a node needs to send a message to another node, it has a route to that node in its routing table
   - Two examples of proactive protocols
     (a) Destination-sequenced distance vector (DSDV)
     (b) Optimised link state routing (OSLR)

2. Reactive

   - Called on-demand routing protocols
   - Does not maintain routing tables at all times
   - A route is discovered when it is needed, i.e. when the source has some data to send
   - Two examples of reactive protocols
     (a) Ad-hoc on demand distance vector (AODV)
     (b) Dynamic source routing (DSR)

3. Combination of proactive and reactive

   - Zone Routing Protocol (ZRP)

4. Local connectivity management

   - MANET protocols have in common, that they need to have a mechanism that allows discovery of neighbours
   - Neighbours are nodes within broadcast range, i.e. they can be reached within one hop
   - Neighbours can be found by periodically broadcasting "Hello" messages. These won't be relayed. These messages contain information about the neighbours known by the sending node.
   - When a hello message from x is received by y, y can check, if y is in the neighbour list of x. If y is there, the link must be bi-directional. Otherwise, it's likely uni-directional.

5. Destination-Sequenced Distance Vector

   - Uses sequence numbers to avoid loops.
   - Has message overhead which grows as $O(n^2)$, when a change occurs in the network.

   (a) Using sequence numbers
     - Each node maintains a counter that represents its current sequence number. This counter starts at zero and is incremented by two whenever it is updated.
     - A sequence number set by a node itself, will always be even.
     - The number of a node is propagared through the network in the update messages that are sent to the neighbours.

- Whenever an update message is sent, the sender increments its number and prefixes this to the message.
- Whenever an update message is received, the receiver can get the number. This information is stored in the receiving nodes route table and is further propagated in the network in subsequent update messages sent, regarding routes to that destination.
- Like this, the sequence number set by the destination node is stamped on every route to that node.
- Update messages thus contain; a destination node, a cost of the route, a next-hop node and the latest known destination sequence number.
- On receiving an update message, these rules apply:
  i. If the sequence number of the updated route is higher than what is currently stored, the route table is updated.
  ii. If the numbers are the same, the route with the lowest cost is chosen.
- If a link break is noticed, the noticer sets the cost to be inf to that node and increments the gone node's sequence number by one and sends out an update.
- Thus, the sequence number is odd, whenever a node discovers a link breakage.
- Because of this, any further updates from the disappeared node, will automatically supersede this number
- This makes DSDV loop free
- Sequence numbers are changed in the following ways
  i. When a line breaks. The number is changed by a neighbouring node. Link breakage can't form a loop
  ii. When a node sends an update message. The node changes its own sequence number and broadcasts this. This information is passed on from the neighbours.
- Thus, the closer you are, the more recent sequence number you know.
- When picking routes, we trust the routers who knows the most recent sequence number, in addition to picking the shortest route.

(b) Sending updates

- Two types of updates; full and incremental
- Full updates contain information about all routes known by the sender. These are sent infrequently.
- Incremental updates contain only changed routes. These are sent regularly.
- Decreases control bandwidth.
- Full updates are sent in some relatively large interval
- Incremental updates are sent frequently
- Full updates are allowed to use multiple network protocol data units, NPDUs (??????), whereas incremental can only use one. Too many incremental to fit in a single -> send full instead
- When an update to a route is received, different actions are taken, depending on the information:
  i. If it's a new route, schedule for immediate update, send incremental update ASAP
  ii. If a route has improved, send in next incremental update
  iii. If sequence number has changed, but route hasn't, send in next incremental if space

(c) Issue

- Suffers from routing fluctuations
- A node could repeatedly switch between a couple of routes
- Essentially, one route is slower, but for some reason the update comes from that first, while the other is quicker, but the number comes slower. You receive an update and update the route to be the slowest. Then you receive the slower update and have to to another update, as the new route is shorter.
- Fixed by introduction delay. If the cost to a destination changes this information is scheduled for advertisement at a time depending on the average settling time for that destination.

6. Optimised Link State Routing

- Designed to be effective in an environment with a dense population of mobile devices, which communicate often.
- Introduces multi point relay (MPR) sets. These are a subset of one-hop neighbours of a node, that is used for routing the messages of that node. These routers are called MPR selectors.

(a) Multipoint relay set

- Selected independently by each node as a subset of its neighbours.
- Selected such that the set covers all nodes that are two hops away
- Doesn't have to be optimal
- Each node stores a list of both one-hop and two-hop neighbours. Collected from the hello messages which are broadcasted regardless. These should also contain neighbours. This means that all neighbours of the one-hop neighbours, must be the set of two-hop neighbours. We can then simply check if we know all.

(b) Routing with MPR

- A topology control (TC) message is required to create a routing table for the entire network
- This is sent via the MPR and will eventually reach the entire network. It's not as much flooding as the standard LS algorithm.

7. Ad-hoc On-Demand Distance Vector

- Reactive
- Routes are acquired when they are needed
- Assumes symmetrical links
- Uses sequence numbers to avoid loops

(a) Path Discovery

- When a node wishes to send something, a path discovery mechanism is triggered
- If node x wishes to send something to node y, but it doesn't know a route to y, a route request (RREQ) message is send to x's neighbours. The RREG contains:
  i. Source address
  ii. Source seq no
  iii. Broadcast id - A unique id of the current RREQ
  iv. Destination addr
  v. Destionation seq no

vi. Hop count - The number of hops so far, incremented when RREQ is forwarded

- (source addr, broadcast id) uniquely identifies a RREQ. This can be used to check if RREG has been seen before.
- When RREQ is received, two actions can be taken
  i. If a route to the destination is known and that path has a sequence number equal or greater than the destionation seq no in the RREQ, it responds to the RREQ by sending a RREP (route reply) back to the source.
  ii. If it doesn't have a recent route, it broadcasts the RREQ to neighbours with an increased hop count.
- When a RREQ is received, the address of the neighbour from whom this was received, is recorded. This allows for the generation of a reverse path, should the destination node be found.
- RREP contains source, destination addr, destionation seq no, the total number of hops from source to dest and a lifetime value for the route.
- If multiple RREPs are received by an intermediary node, only the first one is forwarded and the rest are forwarded if their destination sequence number is higher or they have a lower hop count, but the same dest seq no.
- When the RREP is send back to the source, the intermediary nodes record which node they received the RREP from, to generate a forward path to route data along.

(b) Evaluation

- Tries to minimise control traffic flowing, by having nodes only maintain active routes.
- Loops prevented with sequence numbers
- No system wide broadcasts of entire routing tables
- Every route is only maintained, as long as it's used. It has a timeout and is discarded, if this timeout is reached.
- Path finding can be costly, as a long of RREG gets propagated through the network
- Expanding ring algorithm can help control the amount of messages going out, but if the receiver isn't close, this can be even more costly than the standard way

- Upon link failure; Upstream neighbour sends RREP with seq. no. +1 and hop count set to infinity to any active neighbours—that is neighbours that are using the route.

8. Dynamic Source Routing

   - On-demand protocol
   - DSR is a source routing protocol. This is main difference between DSR and AODV
   - Source routing is a technique, where every message contains a header describing the entire path that the message must follow.
   - When a message is received, the node checks if it's the destination node, if not, it forwards the message to the next node in the path.
   - There is no need for intermediate nodes to keep any state about active routes, as was the case in the AODV protocol.
   - DSR doesn't assume symmetrical links and can use uni-directional links, i.e. one route can be used from A to B and then a different route from B to A.

   (a) Path Discovery

      - Discovery is similiar to AODV
      - RREQ contains the source and destination address and a request id.
      - Source address and request id defines the RREQ
      - When an intermediate node receives a RREQ it does a few things.
         i. If it has no route to the dest, it appends itself to the list of nodes in the RREQ and then forwards it to its neighbours
         ii. If it does have a route to the dest, it appends this route to the list of nodes and sends a RREP back to the source, containing this route.
      - This system uses the same amount of messages, as AODV, and finds the same routes.
      - When a node is ready to send RREP back to source, it can do one of 3 things:
         i. If it already has a route to the source, it can send RREP back along this path

ii. It can reverse the route in the RREP (i.e., the list the nodes append themselves to, when forwarding)

iii. It can initiate a new RREQ to find a route to the source

- The second option assumes symmetrical links.
- The third approach can cause a loop, as the source and the dest host can endlessly look for each other
- Can be avoided by piggybacking the RREP on the second RREQ message. The receiver of this RREQ will be given a path to use when returning the reply.

(b) Route cache

- There is no route table
- DSR use a route cache of currently known routes. The route cache of a node is in effect a tree rooted at the node
- This tree can contain multiple routes to a single destination
- This means it's most robust against broken links, as even though a link breaks, another can maybe be used
- Might take up $O(n^2)$ space

(c) Promiscuous mode operation

- DSR takes advantage of the fact that wireless devices can overhear messages that aren't addressed to them.
- Since messages tend to be broadcasted, other nodes within the range of the broadcast, can also read the message
- Having nodes overhear messages that are not addressed to them, is called promiscuous mode operation.
- It's not required for DSR to work, but it improves the protocol.
- When two nodes on a path moves out of transmission range, some sort of acking mechanism must be used. This is usually done by using link-layer acks, but if such functionality isn't available, this must be done through other means.
- A passive ack is when a host, after sending a message to the next hop host in a path, overhears that the receiving host is transmitting the message again. This can be taken as a sign, that the host has in fact received the message and is now in the process of forwarding it towards the next hop.
- A host that overhears a message may add the route of the message to its route cache

- It might also be an error message, then the route cache can be corrected.
- Can also be used for route shortening, if A sends to B who sends to C, but C overhears the message to B, C can send an RREP to A and let A know the route can be shortened.

(d) Evaluation

- Like AODV, DSR only uses active routes, i.e. routes timeout
- Control messages used are kept low by using same optimisations as AODV
- Storage overhead is O(n) - Route cache and information about recently received RREQ
- Loops are easily avoided in source routing, since nodes can just check if they're already a part of a path. If so, message is discarded.

9. Zone Routing Protocol

- Hybrid protocol
- In ZRP, each node defines a zone consisting of all of it's n-hop neighbours, where n may be varied.
- Within this zone, the node proactively maintains a routing table of routes to all other nodes in the zone. This is done using intrazone routing protocol, which is LS based.
- These zones can be used, when sending to nodes within the zone
- Outside the zone, a re-active interzone routing scheme is used.
- This uses a concept called bordercasting.
- The source node sends a route request (essentially an RREQ message) to all of the nodes on the border of its zone.
- These border nodes check if they can reach the dest directly. If not, they propagate the message to their border nodes.

(a) Evaluation

- Less control traffic when doing route discovery, as messages are either sent to border nodes (skipping a lot of intermediary hops) or they're sent directly to someone within the zone.
- More control messages within limited range of the zones though.

- Storage complexity of $O(n^2)$ where n is the number of neighbours within the zone.
- Since LS is used, the running time is $O(m + n \log n)$, where m is edges connecting the n nodes in the zone.
- In dense scenarios, ZRP won't be feasible.

2.2 Energy Efficient MANET Routing

- All mentioned protocols in chapter 2 try to minimise control traffic, which, albeit does save energy since transmitting fewer messages is nice, but this is done primarily to avoid wasting bandwidth.

2.2.1 Introduction to energy efficient routing

- Two main approaches

  1. Power-save
  2. power-control

- Power-save is concerned with sleep states. In a power-save protocol the mobile nodes utilise that their network interfaces can enter into a sleep state where less energy is consumed.

- Power-control utilises no sleep states. Instead the power used when transmitting data is varied; which also varies transmission range of nodes.

- Power-control can save some energy, but the real energy saver is in power-save, as the real waste in most MANETs is idle time.

- As such, power-save is the most important, but power-control can be used to complement it.

- Goal of the energy efficiency is important to define:

- One approach is to maximise overall lifetime of the entire network

- Stronger nodes that have a longer battery life, may be asked to do a lot of the heavy lifting.

- Another approach is to use minimum energy when routing, such that the route using the minimum amount of energy is taken.

- The physical position of nodes can be important when making routing decisions.
- Protocols tend to assume there is some positioning mechanism available, such as GPS.
- This is not assumed here.
- A third energy saving approach is load balancing. The protocol attempts to balance the load in such a way that it maximises overall lifetime. (This sounds a lot like having a few strong nodes do heavylifting)

2.2.2 The power-control approach

- Power-control protocols cut down on energy consumption by controlling the transmission power of the wireless interfaces.
- Turning down transmission power when sending to neighbours is nice. It consumes less energy for the sender, since the range is lowered, less nodes have to spend energy overhearing the message.
- There is a non-linear relation between transmission range and energy used, thus, more hops might in fact yield less energy spent.
- System called PARO uses this, as it allows more intermediary nodes, if this lowers the overall cost of the path.

2.2.3 Power-save approach

- Protocols that use the power-save approach cut down on energy consumption by utilising the sleep states of the network interfaces
- When a node sleeps, it can't participate in the network
- This means these protocols have to either
    1. use retransmissions of messages to make sure that a message is received
    2. make sure that all of the nodes do not sleep at the same time, and thus delegate the work of routing data to the nodes that are awake.

- Power-save protocols define ways in which nodes can take turns sleeping and being awake, so that none, or at least a very small percentage of the messages sent in the network are lost, due to nodes being in the sleep state.
- They are specifications of how it is possible to maximise the amount of time that nodes are sleeping, while still retaining the same connectivity and loss rates comparable to a network where no nodes are sleeping.
- IEEE 802.11 ad hoc power saving mode, part of the IEEE standard, uses sleep states.
- It uses the protocol on the link layer and is as such independent of which routing protocol is used on network layer.
- BECA/AFECA uses retransmissions
- Span specifices when nodes can sleep and delegates routing to the rest

1. IEEE
    - Beacon interval within which each node can take a number of actions
    - In the end of each beacon interval, the nodes compete for transmission of the next beacon, the one who first transmits, win.
    - In the beginning of ea h bea on interval all nodes must be awake.
    - It works in a few phases, where nodes can announce to receivers that they want to send stuff. After this phase, any node which wasn't contacted, can safely sleep.
2. BECA/AFECA
    - The difference between BECA and AFECA is that AFECA takes node density into consideration when determining the period of time that a node may sleep.
    - Both approaches are only power saving algorithms and not routing protocols. This means that they need to work together with some existing MANET routing protocol.
    - It makes sense to choose an on-demand routing protocol for this purpose, as pro-active would keep the nodes alive.

    (a) Basic Energy-Conserving algorithm (BECA)

- Based on retransmissions
- Consists of timing information that defines the periods that nodes spend in the different states defined by the algorithm, and a specification of how many retransmissions are needed.
- BECA has three states
  i. sleeping
  ii. listening
  iii. active
- Some rules to ensure no messages are lost
  i. $T_{listen} = T_{retransmissions}$
  ii. $T_{sleep} = k * T_{retransmissions}$, for some k
  iii. $Number_{of retrans} >= k + 1$
  iv. $T_{idle} = T_{retransmissions}$
- If A sends to B, but B sleeps, the message will be retrans R $>= k + 1$ times with interval $T_{restrans}$, until the message has been received.
- Since T_{sleep} is defined as $k * T_{retrans}$, at least one of the retrans will be received, even when B sleeps just before A transmits the message.
- Incurs higher latency, worst case $k * T_{retrans}$ and on average $(k * T_{retrans}) / 2$. This latency is added for each hop.
- Thus, to keep this low, k must be somewhat small, which counteracts the energy saving.
- Thus, one needs to find a nice ratio.
- Apparently $k = 1$ is nice.
- A nice feature of BECA, which also applies to AFECA, is that in high traffic scenarios, where all nodes are on at all times, nodes are simply kept in the active state. In this way the power saving mechanism is disabled and the performance of the protocol is thus as good as the underlying protocol.

(b) Adaptive Fidelity energy-conserving algorithm (AFECA)

- Same power save model as BECA, except instead of T_{sleep}, it has $T_{variasleep}$
- T_{vs} is varied according to amount of neighbours surrounding a node.
- This is estimated when in listening state, according to how many are overheard.

- Nodes are removed from the estimation after they timeout at T_{gone} time.
- T_{vs} is then defined as T_{vs} = Random(1, $amount_{ofneighbours}$) * T_{sleep}
- Sleep time of (N * T_{sleep}) / 2 on average
- Favours nodes in dense areas, due to N, which is $amount_{ofneighbours}$.
- When $number_{ofretrans}$ isn't changed, but the sleep time is, packets might be lost. A fix could be to make this variable as well.
- Apparently doubles the overall lifetime, as network density rises.

3. Span

3 Accessing and Developing WoT

3.1 Chapter 6

3.1.1 REST STUFF

- The first layer is called access. This layer is aptly named Access because it covers the most fundamental piece of the WoT puzzle: how to connect a Thing to the web so that it can be accessed using standard web tools and libraries.

- REST provides a set of architectural constraints that, when applied as a whole, empha- sizes scalability of component interactions, generality of interfaces, independent deploy- ment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

- In short, if the architecture of any distributed system follows the REST constraints, that system is said to be RESTful.

- Maximises interoperability and scalability

- Five constraints: Client/server, Uniform interfaces, Stateless, Cacheable, Layered system

1. Client/server

- Maximises decoupling, as client doesn't need to know how the server works and vice versa
- Such a separation of concerns between data, control logic, and presentation improves scalability and portability because loose coupling means each component can exist and evolve independently.

2. Uniform interfaces

   - Loose coupling between components can be achieved only when using a uniform interface that all components in the system respect.
   - This is also essential for the Web of Things because new, unknown devices can be added to and removed from the system at any time, and interacting with them will require min- imal effort.

3. Stateless

   - The client context and state should be kept only on the client, not on the server.
   - Each request to server should contain client state, visibility (monitoring and debugging of the server), robustness (recovering from network or application failures) and scalability are improved.

4. Cacheable

   - Caching is a key element in the performance (loading time) of the web today and therefore its usability.
   - Servers can define policies as when data expires and when updates must be reloaded from the server.

5. Layered

   - For example, in order to scale, you may make use of a proxy behaving like a load balancer. The sole purpose of the proxy would then be to forward incoming requests to the appropriate server instance.
   - Another layer may behave like a gateway, and translate HTTP requests to other protocols.
   - Similarly, there may be another layer in the architecture responsible for caching responses in order to minimize the work needed to be done by the server.

6. HATEOAS

    - Servers shouldn't keep track of each client's state because stateless applications are easier to scale. Instead, application state should be addressable via its own URL, and each resource should contain links and information about what operations are possible in each state and how to navigate across states. HATEOAS is particularly useful at the Find layer

7. Principles of the uniform interface of the web

    - Our point here is that what REST and HTTP have done for the web, they can also do for the Web of Things. As long as a Thing follows the same rules as the rest of the web—that is, shares this uniform interface—that Thing is truly part of the web. In the end, the goal of the Web of Things is this: make it possible for any physical object to be accessed via the same uniform interface as the rest of the web. This is exactly what the Access layer enables
    - Addressable resources—A resource is any concept or piece of data in an application that needs to be referenced or used. Every resource must have a unique identi- fier and should be addressable using a unique referencing mechanism. On the web, this is done by assigning every resource a unique URL.
    - Manipulation of resources through representations—Clients interact with services using multiple representations of their resources. Those representations include HTML, which is used for browsing and viewing content on the web, and JSON, which is better for machine-readable content.
    - Self-descriptive messages—Clients must use only the methods provided by the pro- tocol—GET, POST, PUT, DELETE, and HEAD among others—and stick to their meaning as closely as possible. Responses to those operations must use only well-known response codes—HTTP status codes, such as 200, 302, 404, and 500.
    - Hypermedia as the engine of the application state (HATEOAS)—Servers shouldn't keep track of each client's state because stateless applications are easier to scale. Instead, application state should be addressable via its own URL, and each resource should contain links and information about what operations are possi- ble in each state and how to navigate across states.

(a) Principle #1, adressable resources

- REST is a resource-oriented architecture (ROA)
- A resource is explicitly identified and can be individually addressed, by its URI
- A URI is a sequence of characters that unambiguously identifies an abstract or physi- cal resource. There are many possible types of URIs, but the ones we care about here are those used by HTTP to both identify and locate on a network a resource on the web, which is called the URL (Uniform Resource Locator) for that resource.
- An important and powerful consequence of this is the addressability and portability of resource identifiers: they become unique (internet- or intranet-wide)
- Hierachical naming!

(b) Principle #2, manipulation of resources through representation

- On the web, Multipurpose Internet Mail Extensions (MIME) types have been introduced as standards to describe various data for- mats transmitted over the internet, such as images, video, or audio. The MIME type for an image encoded as PNG is expressed with image/png and an MP3 audio file with audio/mp3. The Internet Assigned Numbers Authority (IANA) maintains the list of the all the official MIME media types.
- The tangible instance of a resource is called a representation, which is a standard encoding of a resource using a MIME type.
- HTTP defines a simple mechanism called content negotiation that allows clients to request a preferred data format they want to receive from a specific service. Using the Accept header, clients can specify the format of the representation they want to receive as a response. Likewise, servers specify the format of the data they return using the Content-Type header.
- The Accept: header of an HTTP request can also contain not just one but a weighted list of media types the client understands
- MessagePack can be used to pack JSON into a binary format, to make it lighter.

- A common way of dealing with unofficial MIME types is to use the x- extension, so if you want your client to ask for MessagePack, use Content-Type: application/x-msgpack.

(c) Principle #3: self-descriptive messages

- REST emphasizes a uniform interface between components to reduce coupling between operations and their implementation. This requires every resource to support a standard, common set of operations with clearly defined semantics and behavior.
- The most commonly used among them are GET, POST, PUT, DELETE, and HEAD. Although it seems that you could do everything with just GET and POST, it's important to correctly use all four verbs to avoid bad surprises in your applications or introducing security risks.
- CRUD operations; create, read, update and delete
- HEAD is a GET, but only returns the headers
- POST should be used only to create a new instance of something that doesn't have its own URL yet
- PUT is usually modeled as an idempotent but unsafe update method. You should use PUT to update something that already exists and has its own URL, but not to create a new resource
- Unlike POST, it's idempotent because sending the same PUT message once or 10 times will have the same effect, whereas a POST would create 10 different resources.
- A bunch of error codes as well: 200, 201, 202, 401, 404, 500, 501
- CORS—ENABLING CLIENT-SIDE JAVASCRIPT TO ACCESS RESOURCES

(d) CORS

- Although accessing web resources from different origins located on various servers in any server-side application doesn't pose any problem, JavaScript applications running in web browsers can't easily access resources across origins for security reasons. What we mean by this is that a bit of client-side JavaScript code loaded from the domain apples.com won't be allowed by the browser to retrieve particular representations

of resources from the domain oranges.com using particular verbs.

- This security mechanism is known as the same- origin policy and is there to ensure that a site can't load any scripts from another domain. In particular, it ensures that a site can't misuse cookies to use your credentials to log onto another site.
- Fortunately for us, a new standard mechanism called cross-origin resource sharing (CORS)9 has been developed and is well supported by most modern browsers and web servers.

When a script in the browser wants to make a cross-site request, it needs to include an Origin header containing the origin domain. The server replies with an Access- Control-Allow-Origin header that contains the list of allowed origin domains (or * to allow all origin domains)

- When the browser receives the reply, it will check to see if the Access-Control- Allow-Origin corresponds to the origin, and if it does, it will allow the cross-site request.

For verbs other than GET/HEAD, or when using POST with representations other than application/x-www-form-urlencoded, multipart/form-data, or text/ plain, an additional request called preflight is needed. A preflight request is an HTTP request with the verb OPTIONS that's used by a browser to ask the target server whether it's safe to send the cross-origin request.

(e) Principle #4 : Hypermedia as the Engine of Application State

- contains two subconcepts: hypermedia and application state.
- This fourth principle is centered on the notion of hypermedia, the idea of using links as connections between related ideas.
- Links have become highly popular thanks to web browsers yet are by no means limited to human use. For example, UUIDs used to identify RFID tags are also links.
- Based on this representation of the device, you can easily follow these links to retrieve additional information about the subresources of the device
- The application state—the AS in HATEOAS—refers to a step in a process or workflow, similar to a state machine, and REST requires the engine of application state to be hypermedia driven.

- Each possible state of your device or application needs to be a RESTful resource with its own unique URL, where any client can retrieve a representation of the current state and also the possible transitions to other states. Resource state, such as the status of an LED, is kept on the server and each request is answered with a representation of the current state and with the necessary information on how to change the resource state, such as turn off the LED or open the garage door.
- In other words, applications can be stateful as long as client state is not kept on the server and state changes within an application happen by following links, which meets the self-contained-messages constraint.
- The OPTIONS verb can be used to retrieve the list of operations permitted by a resource, as well as metadata about invocations on this resource.

(f) Five-step process

- A RESTful architecture makes it possible to use HTTP as a universal protocol for web-connected devices. We described the process of web-enabling Things, which are summarized in the five main steps of the web Things design process:
- Integration strategy—Choose a pattern to integrate Things to the internet and the web, either directly or through a proxy or gateway. This will be covered in chapter 7, so we'll skip this step for now.
- Resource design—Identify the functionality or services of a Thing and organize the hierarchy of these services. This is where we apply design rule #1: address- able resources.
- Representation design—Decide which representations will be served for each resource. The right representation will be selected by the clients, thanks to design rule #2: content negotiation.
- Interface design—Decide which commands are possible for each service, along with which error codes. Here we apply design rule #3: self-descriptive messages.
- Resource linking design—Decide how the different resources are linked to each other and especially how to expose those resources and links, along with the operations and parame-

ters they can use. In this final step we use design rule #4: Hypermedia as the Engine of Application State.

8. Design rules

   (a) #2–CONTENT NEGOTIATION

   - Web Things must support JSON as their default representation.
   - Web Things support UTF8 encoding for requests and responses
   - Web Things may offer an HTML interface/representation (UI).

   (b) #3 : Self-descriptive messages

   - Web Things must support the GET, POST, PUT, and DELETE HTTP verbs.
   - Web Things must implement HTTP status codes 20x, 40x, 50x.
   - Web Things must support a GET on their root URL.
   - Web Things should support CORS

   (c) #4 : HATEOAS

   - Web Things should support browsability with links.
   - Web Things may support OPTIONS for each of its resources.

3.1.2 EVENT STUFF

1. Events and stuff

   - Unfortunately, the request-response model is insufficient for a number of IoT use cases. More precisely, it doesn't match event-driven use cases where events must be communicated (pushed) to the clients as they happen.
   - A client-initiated model isn't practical for applications where notifications need to be sent asynchronously by a device to clients as soon as they're produced.
   - polling is one way of circumventing the problem, however it's inefficient, as the client will need to make many requests which will simply return the same response. Additionally, we might not "poll" at the exact time an event takes place.

- Most of the requests will end up with empty responses (304 Not Modified) or with the same response as long as the value observed remains unchanged.

2. Publish/subscribe

   - What's really needed on top of the request-response pattern is a model called publish/subscribe (pub/sub) that allows further decoupling between data consumers (subscribers) and producers (publishers). Publishers send messages to a central server, called a broker, that handles the routing and distribution of the messages to the various subscribers, depending on the type or content of messages.
   - A publisher can send notifications into a topic, which subscribers can have subscribed to

3. Webhooks

   - The simplest way to implement a publish-subscribe system over HTTP without break- ing the REST model is to treat every entity as both a client and a server. This way, both web Things and web applications can act as HTTP clients by initiating requests to other servers, and they can host a server that can respond to other requests at the same time. This pattern is called webhooks or HTTP callbacks and has become popular on the web for enabling different servers to talk to each other.
   - The implementation of this model is fairly simple. All we need is to implement a REST API on both the Thing and on the client, which then becomes a server as well. This means that when the Thing has an update, it POSTs it via HTTP to the client
   - Webhooks are a conceptually simple way to implement bidirectional communication between clients and servers by turning everything into a server.
   - webhooks have one big drawback: because they need the subscriber to have an HTTP server to push the notification, this works only when the subscriber has a publicly accessible URL or IP address.

4. Comet

- Comet is an umbrella term that refers to a range of techniques for circumventing the limitations of HTTP polling and webhooks by introducing event-based communication over HTTP.
- This model enables web servers to push data back to the browser without the client requesting it explicitly. Since browsers were initially not designed with server-sent events in mind, web application developers have exploited several specification loop- holes to implement Comet-like behavior, each with different benefits and drawbacks.
- Among them is a technique called long polling
- With long poll- ing, a client sends a standard HTTP request to the server, but instead of receiving the response right away, the server holds the request until an event is received from the sensor, which is then injected into the response returned to the client's request that was held idle. As soon as the client receives the response, it immediately sends a new request for an update, which will be held until the next update comes from the sensor, and so on.

5. Websockets

    - WebSocket is part of the HTML5 specification. The increasing support for HTML5 in most recent web and mobile web browsers means WebSocket is becoming ubiquitously available to all web apps
    - WebSockets enables a full-duplex communication channel over a single TCP connection. In plain English, this means that it creates a permanent link between the client and the server that both the client and the server can use to send messages to each other. Unlike techniques we've seen before, such as Comet, WebSocket is standard and opens a TCP socket. This means it doesn't need to encapsulate custom, non-web content in HTTP messages or keep the connection artificially alive as is needed with Comet implementations.
    - A websockets starts out with a handshake: The first step is to send an HTTP call to the server with a special header asking for the protocol to be upgraded to WebSockets. If the web server sup- ports WebSockets, it will reply with a 101 Switch- ing Protocols status code, acknowledging the opening of a full-duplex TCP socket.

- Once the initial handshake takes place, the client and the server will be able to send messages back and forth over the open TCP connection; these messages are not HTTP messages but WebSockets data frames
- The overhead of each WebSockets data frame is 2 bytes, which is small compared to the 871-byte overhead of an HTTP message meta- data (headers and the like)
- the hierarchical structure of Things and their resources as URLs can be reused as-is for WebSockets.
- we can subscribe to events for a Thing's resource by using its corre- sponding URL and asking for a protocol upgrade to WebSockets. Moreover, Web- Sockets do not dictate the format of messages that are sent back and forth. This means we can happily use JSON and give messages the structure and semantics we want.
- Moreover, because WebSockets consist of an initial handshake followed by basic message framing layered over TCP, they can be directly implemented on many plat- forms supporting TCP/IP—not just web browsers. They can also be used to wrap sev- eral other internet-compatible protocols to make them web-compatible. One example is MQTT, a well-known pub/sub protocol for the IoT that can be inte- grated to the web of browsers via WebSockets
- The drawback, however, is that keeping a TCP connection permanently open can lead to an increase in battery consumption and is harder to scale than HTTP on the server side.

6. HTTP/2

- This new version of HTTP allows multiplexing responses—that is, sending responses in parallel, This fixes the head-of-line blocking problem of HTTP/1.x where only one request can be outstanding on a TCP/IP connection at a time.
- HTTP/2 also introduces compressed headers using an efficient and low-memory compression format.
- Finally, HTTP/2 introduces the notion of server push. Concretely, this means that the server can provide content to clients without having to wait for them to send a request. In the long run,

widespread adoption of server push over HTTP/2 might even remove the need for an additional protocol for push like WebSocket or webhooks.

3.1.3 SUMMARY

- When applied correctly, the REST architecture is an excellent substrate on which to create large-scale and flexible distributed systems.
- REST APIs are interesting and easily applicable to enable access to data and ser- vices of physical objects and other devices.
- Various mechanisms, such as content negotiation and caching of Hypermedia as the Engine of Application State (HATEOAS), can help in creating great APIs for Things.
- A five-step design process (integration strategy, resource design, representation design, interface design, and resource linking) allows anyone to create a mean- ingful REST API for Things based on industry best practices.
- The latest developments in the real-time web, such as WebSockets, allow creat- ing highly scalable, distributed, and heterogeneous real-time data processing applications. Devices that speak directly to the web can easily use web-based push messaging to stream their sensor data efficiently.
- HTTP/2 will bring a number of interesting optimizations for Things, such as multiplexing and compression.

3.2 Chapter 7

3.2.1 Connecting to the web

1. Direct Integration
    - The most straightforward integration pattern is the direct integration pattern. It can be used for devices that support HTTP and TCP/IP and can therefore expose a web API directly. This pattern is particularly useful when a device can directly connect to the internet; for example, it uses Wi-Fi or Ethernet
2. Gateway Integration

- Second, we explore the gateway integra- tion pattern, where resource-constrained devices can use non-web protocols to talk to a more powerful device (the gateway), which then exposes a REST API for those non-web devices. This pattern is particularly useful for devices that can't connect directly to the internet; for example, they support only Bluetooth or ZigBee or they have limited resources and can't serve HTTP requests directly.

3. Cloud Integration

   - Third, the cloud integration pattern allows a powerful and scalable web platform to act as a gateway. This is useful for any device that can connect to a cloud server over the internet, regardless of whether it uses HTTP or not, and that needs more capability than it would be able to offer alone.

3.2.2 Five step process

1. Integration strategy—Choose a pattern to integrate Things to the internet and the web. The patterns are presented in this chapter.

2. Resource design—Identify the functionality or services of a Thing, and organize the hierarchy of these services.

3. Representation design—Decide which representations will be served for each resource.

4. Interface design—Decide which commands are possible for each service, along with which error codes.

5. Resource linking design—Decide how the different resources are linked to each other.

1. Direct integration

   - the direct integration pattern is the perfect choice when the device isn't battery powered and when direct access from clients such as mobile web apps is required.
   - the resource design. You first need to consider the physical resources on your device and map them into REST resources.
   - The next step of the design process is the representation design. REST is agnostic of a par- ticular format or representation of the

data. We mentioned that JSON is a must to guarantee interoperability, but it isn't the only interesting data representation available.

- a modular way based on the middleware pattern.
- In essence, a middleware can execute code that changes the request or response objects and can then decide to respond to the client or call the next middleware in the stack using the next() function.
- The core of this implementation is using the Object.observe() function.9 This allows you to asynchronously observe the changes happening to an object by registering a callback to be invoked whenever a change in the observed object is detected.

2. Gateway integration pattern

    - Gateway integration pattern. In this case, the web Thing can't directly offer a web API because the device might not support HTTP directly. An application gateway is working as a proxy for the Thing by offering a web API in the Thing's name. This API could be hosted on the router in the case of Bluetooth or on another device that exposes the web Thing API; for example, via CoAP.
    - The direct integration pattern worked well because your Pi was not battery powered, had access to a decent bandwidth (Wi-Fi/Ethernet), and had more than enough RAM and storage for Node. But not all devices are so lucky. Native sup- port for HTTP/WS or even TCP/IP isn't always possible or even desirable. For batterypowered devices, Wi-Fi or Ethernet is often too much of a power drag, so they need to rely on low-power protocols such as ZigBee or Bluetooth instead. Does it mean those devices can't be part of the Web of Things? Certainly not.
    - Such devices can also be part of the Web of Things as long as there' s an intermedi- ary somewhere that can expose the device's functionality through a WoT API like the one we described previously. These intermediaries are called application gateways (we'll call them WoT gateways hereafter), and they can talk to Things using any non-web application protocols and then translate those into a clean REST WoT API that any HTTP client can use.

- They can add a layer of security or authentication, aggregate and store data temporarily, expose semantic descriptions for Things that don't have any, and so on.
- CoAP is a service layer protocol that is intended for use in resource-constrained internet devices, such as wireless sensor network nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web
- CoAP is an interesting protocol based on REST, but because it isn't HTTP and uses UDP instead of TCP, a gateway that translates CoAP messages from/to HTTP is needed
- It's therefore ideal for device-to-device communi- cation over low-power radio communication, but you can't talk to a CoAP device from a JavaScript application in your browser without installing a special plugin or browser extension. Let's fix this by using your Pi as a WoT gateway to CoAP devices.
- By proxying, the gateway essentially just send a request to the CoAP device whenever the gateway receives a request and it'll return the value to the requester, once it receives a value from the CoAP device.

(a) Summary

- For some devices, it might not make sense to support HTTP or WebSockets directly, or it might not even be possible, such as when they have very limited resources like memory or processing, when they can't connect to the internet directly (such as your Bluetooth activity tracker), or when they're battery-powered. Those devices will use more optimized communication or application protocols and thus will need to rely on a more powerful gateway that connects them to the Web of Things, such as your mobile phone to upload the data from your Bluetooth bracelet, by bridging/translating various protocols. Here we implemented a simple gateway from scratch using Express, but you could also use other open source alternatives such as OpenHab13 or The Thing System.

3. Cloud Integration pattern

- Cloud integration pattern. In this pattern, the Thing can't directly offer a Web API. But a cloud service acts as a powerful

application gateway, offering many more features in the name of the Thing. In this particular example, the web Thing connects via MQTT to a cloud service, which exposes the web Thing API via HTTP and the WebSockets API. Cloud services can also offer many additional features such as unlimited data storage, user management, data visualization, stream processing, support for many concurrent requests, and more.

- Using a cloud server has several advantages. First, because it doesn't have the physical constraints of devices and gateways, it's much more scalable and can process and store a virtually unlimited amount of data. This also allows a cloud platform to support many protocols at the same time, handle protocol translation efficiently, and act as a scalable intermediary that can support many more concurrent clients than an IoT device could.
- Second, those platforms can have many features that might take consid- erable time to build from scratch, from industry-grade security, to specialized analytics capabilities, to flexible data visualization tools and user and access management
- Third, because those platforms are natively connected to the web, data and services from your devices can be easily integrated into third-party systems to extend your devices.

3.2.3 Summary

- There are three main integration patterns for connecting Things to the web: direct, gateway, and cloud.
- Regardless of the pattern you choose, you'll have to work through the following steps: resource design, representation design, and interface design.
- Direct integration allows local access to the web API of a Thing. You tried this by building an API for your Pi using the Express Node framework.
- The resource design step in Express was implemented using routes, each route representing the path to the resources of your Pi.
- We used the idea of middleware to implement support for different representa- tions— for example, JSON, MessagePack, and HTML—in the representation design step.

- The interface design step was implemented using HTTP verbs on routes as well as by integrating a WebSockets server using the ws Node module.
- Gateway integration allows integrating Things without web APIs (or not sup- porting web or even internet protocols) to the WoT by providing an API for them. You tried this by integrating a CoAP device via a gateway on your cloud.
- Cloud integration uses servers on the web to act as shadows or proxies for Things. They augment the API of Things with such features as scalability, analy- tics, and security. You tried this by using the EVRYTHNG cloud.

4 Discovery and Security for the Web of Things

4.1 Chapter 8

- Having a single and common data model that all web Things can share would further increase interoperability and ease of integration by making it possible for applications and services to interact without the need to tailor the application manually for each specific device.
- The ability to easily discover and understand any entity of the Web of Things—what it is and what it does—is called findability.
- How to achieve such a level of interoperability—making web Things findable—is the purpose of the second layer
- The goal of the Find layer is to offer a uniform data model that all web Things can use to expose their metadata using only web standards and best practices.
- Metadata means the description of a web Thing, including the URL, name, current location, and status, and of the services it offers, such as sensors, actuators, com- mands, and properties
- this is useful for discovering web Things as they get con- nected to a local network or to the web. Second, it allows applications, services, and other web Things to search for and find new devices without installing a driver for that Thing

4.1.1 Findability problem

- For a Thing to be interacted with using HTTP and WebSocket requests, there are three fundamental problems

    1. How do we know where to send the requests, such as root URL/resources of a web Thing?
    2. How do we know what requests to send and how; for example, verbs and the format of payloads?
    3. How do we know the meaning of requests we send and responses we get, that is, semantics?

- The bootstrap problem. This problem is concerned with how the initial link between two entities on the Web of Things can be established.

- Lets assume the Thing can be found, how is it interacted with, if it exposes a UI at the root of its URL? In this case, a clean and user-centric web interface can solve problem 3 because humans would be able to read and understand how to do this.

- Problem 2 also would be taken care of by the web page, which would hardcode which request to send to which endpoint.

- But what if the heater has no user interface, only a RESTful API?1 Because Lena is an experienced front-end developer and never watches TV, she decides to build a sim- ple JavaScript app to control the heater. Now she faces the second problem: even though she knows the URL of the heater, how can she find out the structure of the heater API? What resources (endpoints) are available? Which verbs can she send to which resource? How can she specify the temperature she wants to set? How does she know if those parameters need to be in Celsius or Fahrenheit degrees?

4.1.2 Discovering Things

- The bootstrap problem deals with two scopes:

    1. first, how to find web Things that are physically nearby—for example, within the same local network
    2. second, how to find web Things that are not in the same local network—for example, find devices over the web.

1. Network discovery

   - In a computer network, the ability to automatically discover new participants is common.
   - In your LAN at home, as soon as a device connects to the network, it automatically gets an IP address using DHCP
   - Once the device has an IP address, it can then broadcast data packets that can be caught by other machines on the same network.
   - a broadcast or multicast of a message means that this message isn't sent to a particular IP address but rather to a group of addresses (multicast) or to everyone (broadcast), which is done over UDP.
   - This announcement process is called a network discovery protocol, and it allows devices and applications to find each other in local networks. This process is commonly used by various discovery protocols such as multicast Domain Name System (mDNS), Digital Living Network Alliance (DLNA), and Universal Plug and Play (UPnP).
   - Most internet-connected TVs and media players can use DLNA to discover network-attached storage (NAS)
   - your laptop can find and configure printers on your network with minimal effort thanks to network-level discovery protocols such as Apple Bonjour that are built into iOS and OSX.

   (a) mDNS

      - In mDNS, clients can discover new devices on a network by listening for mDNS mes- sages such as the one in the following listing. The client populates the local DNS tables as messages come in, so, once discovered, the new service—here a web page of a printer—can be used via its local IP address or via a URI usually ending with the .local domain. In this example, it would be `http://evt-bw-brother.local`.
      - The limitation of mDNS, and of most network-level discovery protocols, is that the network-level information can't be directly accessed from the web.

   (b) Network discovery on the web

- Because HTTP is an Application layer protocol, it doesn't know a thing about what's underneath—the network protocols used to shuffle HTTP requests around.
- The real question here is why the configu- ration and status of a router is only available through a web page for humans and not accessible via a REST API. Put simply, why don't all routers also offer a secure API where its configuration can be seen and changed by others' devices and applications in your network?
- Providing such an API is easy to do. For example, you can install an open-source operating system for routers such as OpenWrt and modify the software to expose the IP addresses assigned by the DHCP server of the router as a JSON document.
- This way, you use the existing HTTP server of your router to create an API that exposes the IP addresses of all the devices in your network. This makes sense because almost all networked devices today, from printers to routers, already come with a web user inter- face. Other devices and applications can then retrieve the list of IP addresses in the network via a simple HTTP call (step 2 in figure 8.3) and then retrieve the metadata of each device in the network by using their IP address (step 3 of figure 8.3).

(c) Resource discovery on the web

- Although network discovery does the job locally, it doesn't propagate beyond the boundaries of local networks.
- how do we find new Things when they connect, how do we understand the services they offer, and can we search for the right Things and their data in composite applications?
- On the web, new resources (pages) are discovered through hyperlinks. Search engines periodically parse all the pages in their database to find outgoing links to other pages. As soon as a link to a page not yet indexed is found, that new page is parsed and added to directory. This process is known as web crawling.

(d) Crawling

- From the root HTML page of the web Thing, the crawler can find the sub-resources, such as sensors and actuators, by

discovering outgoing links and can then create a resource tree of the web Thing and all its resources. The crawler then uses the HTTP OPTIONS method to retrieve all verbs supported for each resource of the web Thing. Finally, the crawler uses content negotiation to understand which format is available for each resource.

(e) HATEOAS and web linking

- The simple way of crawling, of basically looping through links found is a good start, but it also has several limitations. First, all links are treated equally because there's no notion of the nature of a link; the link to the user interface and the link to the actuator resource look the same—they're just URLs.
- Additionally, it requires the web Thing to offer an HTML interface, which might be too heavy for resource-constrained devices. Finally, it also means that a client needs to both understand HTML and JSON to work with our web Things.
- A better solution for discovering the resources of any REST API is to use the HATEOAS principle to describe relationships between the various resources of a web Thing.
- A simple method to implement HATEOAS with REST APIs is to use the mechanism of web linking defined in RFC 5988. The idea is that the response to any HTTP request to a resource always contains a set of links to related resources—for example, the previous, next, or last page that contains the results of a search. These would be contained in the LINK header.
- encoding the links as HTTP headers introduces a more general framework to define relationships between resources outside the representation of the resource—directly at the HTTP level.
- When doing an HTTP GET on any Web Thing, the response should include a Link header that contains links to related resources. In particular, you should be able to get information about the device, its resources (API endpoints), and the documentation of the API using only Link headers.
- The URL of each resource is contained between angle brackets (<URL>) and the type of the link is denoted by rel="X", where X is the type of the rela- tion.

(f) New HATEOAS rel link things

- REL="MODEL" : This is a link to a Web Thing Model resource; see section 8.3.1.
- REL="TYPE" : This is a link to a resource that contains additional metadata about this web Thing.
- REL="HELP" : This relationship type is a link to the documentation, which means that a GET to devices.webofthings.io/help would return the documentation for the API in a human-friendly (HTML) or machine-readable (JSON) format.
- REL="UI" : This relationship type is a link to a graphical user interface (GUI) for interacting with the web Thing.

4.1.3 Describing web Things

- knowing only the root URL is insufficient to interact with the Web Thing API because we still need to solve the sec- ond problem mentioned at the beginning of this chapter: how can an application know which payloads to send to which resources of a web Thing?
- how can we formally describe the API offered by any web Thing?
- The simplest solution is to provide a written documentation for the API of your web Thing so that developers can use it (1 and 2 in figure 8.4).
- This approach, however, is insufficient to automatically find new devices, understand what they are, and what services they offer.
- In addition, manual implementation of the payloads is more error-prone because the developer needs to ensure that all the requests they send are valid
- By using a unique data model to define formally the API of any web Thing (the Web Thing Model), we'll have a powerful basis to describe not only the metadata but also the operations of any web Thing in a standard way (cases 3 and 4 of figure 8.4).
- This is the cornerstone of the Web of Things: creating a model to describe physical Things with the right balance between expressiveness—how flexible the model is—and usability— how easy it is to describe any web Thing with that model.

1. Introducing the Web Thing model

   - Once we find a web Thing and understand its API structure, we still need a method to describe what that device is and does. In other words, we need a conceptual model of a web Thing that can describe the resources of a web Thing using a set of well-known concepts.
   - In the previous chapters, we showed how to organize the resources of a web Thing using the /sensors and /actuators end points. But this works only for devices that actually have sensors and actuators, not for complex objects and scenarios that are com- mon in the real world that can't be mapped to actuators or sensors. To achieve this, the core model of the Web of Things must be easily applicable for any entity in the real world, ranging from packages in a truck, to collectible card games, to orange juice bot- tles. This section provides exactly such a model, which is called the Web Thing Model.

   (a) Entities

      - the Web of Things is composed of web Things.
      - A web Thing is a digital representation of a physical object—a Thing—accessible on the web. Think of it like this: your Facebook profile is a digital representation of yourself, so a web Thing is the "Facebook profile" of a physical object.
      - The web Thing is a web resource that can be hosted directly on the device, if it can connect to the web, or on an intermediate in the network such as a gateway or a cloud service that bridges non-web devices to the web.
      - All web Things should have the following resources:
         i. Model—A web Thing always has a set of metadata that defines various aspects about it such as its name, description, or configurations.
         ii. Properties—A property is a variable of a web Thing. Properties represent the internal state of a web Thing. Clients can subscribe to properties to receive a notification message when specific conditions are met; for example, the value of one or more properties changed.
         iii. Actions—An action is a function offered by a web Thing. Clients can invoke a function on a web Thing by sending

an action to the web Thing. Examples of actions are "open" or "close" for a garage door, "enable" or "disable" for a smoke alarm, and "scan" or "check in" for a bottle of soda or a place. The direc- tion of an action is from the client to the web Thing.

iv. Things—A web Thing can be a gateway to other devices that don't have an inter- net connection. This resource contains all the web Things that are proxied by this web Thing. This is mainly used by clouds or gateways because they can proxy other devices.

i. Metadata
   - In the Web Thing Model, all web Things must have some associated metadata to describe what they are. This is a set of basic fields about a web Thing, including its identifiers, name, description, and tags, and also the set of resources it has, such as the actions and properties. A GET on the root URL of any web Thing always returns the metadata using this format, which is JSON by default

ii. Properties
   - Web Things can also have properties. A property is a collection of data values that relate to some aspect of the web Thing. Typically, you'd use properties to model any dynamic time series of data that a web Thing exposes, such as the current and past states of the web Thing or its sensor values—for example, the temperature or humidity sensor readings.

iii. Actions
   - Actions are another important type of resources of a web Thing because they represent the various commands that can be sent to that web Thing.
   - In theory, you could also use properties to change the status of a web Thing, but this can be a prob- lem when both an application and the web Thing itself want to edit the same property.
   - The actions object of the Web Thing Model has an object called resources, which contains all the types of actions (commands) supported by this web Thing.
   - Actions are sent to a web Thing with a POST to the URL

of the action {WT}/actions/{id}, where id is the ID of the action

iv. Things

- a web Thing can act as a gateway between the web and devices that aren't connected to the internet. In this case, the gateway can expose the resources—properties, actions, and metadata—of those non-web Things using the web Thing.
- The web Thing then acts as an Application-layer gateway for those non-web Things as it converts incoming HTTP requests for the devices into the various protocols or interfaces they support natively. For example, if your WoT Pi has a Bluetooth dongle, it can find and bridge Bluetooth devices nearby and expose them as web Things.
- The resource that contains all the web Things proxied by a web Thing gateway is {WT}/things, and performing a GET on that resource will return the list of all web Things currently available

2. The WoT pie model

- A new tree structure, fitting the discussed model, where the different sensors end up in /properties, setLedState ends up in /actions, we have no /things and /model is the metadata as well as all sensor data, their properties, the actions, everything.
- Following the model allows for dynamically creating routes and such, as all information is maintained in the model of the Thing, /model, /properties, /actions, /things.

3. Summary

- In this section, we introduced the Web Thing Model, a simple JSON-based data model for a web Thing and its resources. We also showed how to implement this model using Node.js and run it on a Raspberry Pi. We showed that this model is quite easy to understand and use, and yet is sufficiently flexible to represent all sorts of devices and products using a set of properties and actions. The goal is to propose a uniform way to describe web Things and their capabilities so that any HTTP client can find web Things and interact with them. This is sufficient for most use cases, and

this model has all you need to be able to generate user interfaces for web Things automatically.

4.1.4 The Semantic Web of Things (Ontologies)

- In an ideal world, search engines and any other applications on the web could also understand the Web Thing Model. Given the root URL of a web Thing, any applica- tion could retrieve its JSON model and understand what the web Thing is and how to interact with it.

- The question now is how to expose the Web Thing Model using an existing web standard so that the resources are described in a way that means some- thing to other clients. The answer lies in the notion of the Semantic Web and, more precisely, the notion of linked data that we introduce in this section.

- Semantic Web refers to an extension of the web that promotes common data formats to facilitate meaningful data exchange between machines. Thanks to a set of stan- dards defined by the World Wide Web Consortium (W3C), web pages can offer a stan- dardized way to express relationships among them so that machines can understand the meaning and content of those pages. In other words, the Semantic Web makes it easier to find, share, reuse, and process information from any content on the web thanks to a common and extensible data description and interchange format.

1. Linked Data and RDFa

   - The HTML specification alone doesn't define a shared vocabulary that allows you to describe in a standard and non-ambiguous manner the elements on a page and what they relate to.

   (a) Linked Data

   - Enter the vision of linked data, which is a set of best practices for publishing and connecting structured data on the web, so that web resources can be interlinked in a way that allows computers to automatically understand the type and data of each resource.
   - This vision has been strongly driven by complex and heavy standards and tools centered on the Resource Description Framework (RDF)

- Although powerful and expressive, RDF would be overkill for most simple scenarios, and this is why a simpler method to structure con- tent on the web is desirable.
- RDFa emerged as a lighter version of RDF that can be embedded into HTML code
- Most search engines can use these annotations to generate better search listings and make it easier to find your websites.
- using RDFa to describe the metadata of a web Thing will make that web Thing findable and search- able by Google.

(b) RFDa

- vocab defines the vocabulary used for that element, in this case the Web of Things Model vocabulary defined previously.
- property defines the various fields of the model such as name, ID, or descrip- tion.
- typeof defines the type of those elements in relation to the vocabulary of the element.
- This allows other applications to parse the HTML representation of the device and automatically understand which resources are available and how they work.

(c) JSON-LD

- JSON-LD is an interesting and lightweight semantic annotation format for linked data that, unlike RDFa and Microdata, is based on JSON.29 It's a simple way to semanti- cally augment JSON documents by adding context information and hyperlinks for describing the semantics of the different elements of a JSON objects.

(d) Micro-summary

- This simple example already illustrates the essence of JSON-LD it gives a context to the content of a JSON document. As a consequence, all clients that understand the `http://schema.org/Product` context will be able to automatically process this informa- tion in a meaningful way. This is the case with search engines, for example. Google and Yahoo! process JSON-LD payloads using the Product schema to render special search results; as soon as it gets indexed, our Pi will be known by Google and Yahoo! as a Raspberry Pi product. This means that the more semantic data we add

to our Pi, the more findable it will become. As an example, try adding a location to your Pi using the Place schema,33 and it will eventually become findable by location.

We could also use this approach to create more specific schemas on top of the Web Thing Model; for instance, an agreed-upon schema for the data and functions a wash- ing machine or smart lock offers. This would facilitate discovery and enable automatic integration with more and more web clients.

4.1.5 Summary

- The ability to find nearby devices and services is essential in the Web of Things and is known as the bootstrap problem. Several protocols can help in discover- ing the root URL of Things, such as mDNS/Bonjour, QR codes or NFC tags.
- The last step of the web Things design process, resource linking design (also known as HATEOAS in REST terms), can be implemented using the web linking mechanism in HTTP headers.
- Beyond finding the root URL and sub-resources, client applications also need a mechanism to discover and understand what data or services a web Thing offers.
- The services of Things can be modeled as properties (variables), actions (func- tions), and links. The Web Thing Model offers a simple, flexible, fully web-com- patible, and extensible data model to describe the details of any web Thing. This model is simple to adapt for your devices and easy to use for your products and applications.
- The Web Thing Model can be extended with more specific semantic descriptions such as those based on JSON-LD and available from the Schema.org repository.

4.2 Chapter 9

- In most cases, Internet of Things deployments involve a group of devices that com- municate with each other or with various applications within closed networks— rarely over open networks such as the internet. It would be fair to call such deploy- ments the "intranets of Things" because they're essentially isolated, private net- works that only a few entities can access. But the real power of the Web of Things lies in

opening up these lonely silos and facilitating interconnection between devices and applications at a large scale.

- when it comes to public data such as data.gov initiatives, real-time traffic/weather/pollution conditions in a city, or a group of sensors deployed in a jungle or a volcano, it would be great to ensure that the general public or researchers anywhere in the world could access that data. This would enable anyone to create new innovative applications with it and possibly gener- ate substantial economic, environmental, and social value.

- How to share this data in secure and flexible way is what Layer 3 provides,

- The Share layer of the Web of Things. This layer focuses on how devices and their resources must be secured so that they can only be accessed by authorized users and applications.

- First, we'll show how Layer 3 of the WoT architecture covers the security of Things: how to ensure that only authorized parties can access a given resource. Then we'll show how to use existing trusted systems to allow sharing physical resources via the web.

4.2.1 Securing Things

- Ultimately, every security breach hurts the entire web because it erodes the overall trust of users in technology.

- Security in the Web of Things is even more critical than in the web. Because web Things are physical objects that will be deployed everywhere in the real world, the risks associated with IoT attacks can be catastrophic.

- Digitally augmented devices allow collecting fine-grained information about people, when they took their last insulin shot, their last jog and where they ran. It can also be used to remote control cars, houses and the like.

- the majority of IoT solutions don't comply with even the most basic security best practices; think clear-text passwords and communications, invalid certificates, old software versions with exploitable bugs, and so on.

1. Securing the IoT has three major problems
    - First, we must consider how to encrypt the communications between two enti- ties (for example, between an app and a web Thing) so that a malicious inter- ceptor—a "man in the middle"—can't access the data being transmitted in clear text. This is referred to as securing the channel
    - Second, we must find a way to ensure that when a client talks to a host, it can ensure that the host is really "himself"
    - Third, we must ensure that the correct access control is in place. We need to set up a method to control which user can access what resource of what server or Thing and when and then to ensure that the user is really who they claim to be.
2. Encryption 101
    - encryption is an essential ingredient for any secure system.
    - Without encryption, any attempt to secure a Thing will be in vain because attackers can sniff the communication and understand the security mechanisms that were put in place.

    (a) Symmetric Encryption
        - The oldest form of encoding a message is symmetric encryption. The idea is that the sender and receiver share a secret key that can be used to both encode and decode a message in a specific way

    (b) Assymetric Encryption
        - another method called asymmetric encryption has become popular because it doesn't require a secret to be shared between parties. This method uses two related keys, one public and the other private (secret)

3. Web Security with TLS: The S of HTTPS
    - Fortunately , there are standard protocols for securely encrypting data between clients and servers on the web.
    - The best known protocol for this is Secure Sockets Layer (SSL)
    - SSL 3.0 has a lot of vulnerabilities (Heartbleed and the like). These events inked the death of this proto- col, which was replaced by the much more secure but conceptually similar Transport Layer Security (TLS)

(a) TLS 101

- Despite its name, TLS is an Application layer protocol (see chapter 5). TLS not only secures HTTP (HTTPS) communication but is also the basis of secure WebSocket (WSS) and secure MQTT (MQTTS)
- First, it helps the client ensure that the server is who it says it is; this is the SSL/TLS authentication. Second, it guarantees that the data sent over the communication channel can't be read by any- one other than the client and the server involved in the transaction (also known as SSL/TLS encryption).
- The client, such as a mobile app, tells the server, such as a web Thing, which protocols and encryption algorithms it supports. This is somewhat similar to the content negotiation process we described in chapter 6.
- The server sends the public part of its certificate to the client. The goal here is for the client to make sure it knows who the server is. All web clients have a list of certificates they trust.12 In the case of your Pi, you can find them in /etc/ssl/certs. SSL certificates form a trust chain, meaning that if a client doesn't trust certificate S1 that the server sends back, but it trusts certificate S2 that was used to sign S1, the web client can accept S1 as well.
- The rest of the process generates a key from the public certificates. This key is then used to encrypt the data going back and forth between the server and the client in a secure manner. Because this process is dynamic, only the client and the server know how to decrypt the data they exchange during this session. This means the data is now securely encrypted: if an attacker manages to capture data packets, they will remain meaningless.

(b) Beyond Self-signed certificates

- Clearly, having to deal with all these security exceptions isn't nice, but these excep- tions exist for a reason: to warn clients that part of the security usually covered by SSL/ TLS can't be guaranteed with the certificate you generated. Basically, although the encryption of messages will work with a self-signed certificate (the one you created with the previous command), the authenticity of the server (the Pi) can't be guaran- teed. In consequence, the chain of trust is broken—problem 2

- In an IoT context, this means that attackers could pretend to be the Thing you think you're talk- ing to.
- The common way to generate certificates that guarantee the authenticity of the server is to get them from a well-known and trusted certificate authority (CA). There exists an amount of these; LetsEncrypt, Symantec and GeoTrust.

4.2.2 Authentication and access control

- Once we encrypt the communication between Things and clients as shown in the pre- vious section, we want to enable only some applications to access it.
- First, this means that the Things—or a gateway to which Things are connected—need to be able to know the sender of each request (identification).
- Second, devices need to trust that the sender really is who they claim to be (authentication)
- Third, the devices also need to know if they should accept or reject each request depending on the identity of this sender and which request has been sent (authorization).

1. Access control with REST and API tokens
    - Server-based authentication is used when we use our username/password to log into a website, we initiate a secure session with the server that's stored for a limited time in the server application's memory or in a local browser cookie.
    - server-based authentication is usually stateful because the state of the client is stored on the server. But as you saw in chapter 6, HTTP is a stateless protocol; therefore, using a server-based authentication method goes against this principle and poses certain problems. First, the performance and scalability of the overall systems are limited because each session must be stored in memory and over- head increases when there are many authenticated users. Second, this authentication method poses certain security risks—for example, cross-site request forgery.
    - alternative method called token-based authentication has become popular and is used by most web APIs.

- Because this token is added to the headers or query parameters of each HTTP request sent to the server, all interactions remain stateless.
- API tokens shouldn't be valid forever. API tokens, just like passwords, should change regularly.

2. OAuth: a web authorization framework

   - OAuth is an open standard for authorization and is essentially a mechanism for a web or mobile app to delegate the authentication of a user to a third-party trusted service; for example, Facebook, LinkedIn, or Google.
   - OAuth dynamically generates access tokens using only web protocols.
   - OUath allows sharing resources and token sharing between applications.
   - In short, OAuth standardizes how to authenticate users, generate tokens with an expiration date, regenerate tokens, and provide access to resources in a secure and standard manner over the web.
   - At the end of the token exchange process, the application will know who the user is and will be able to access resources on the resource server on behalf of the user. The application can then also renew the token before it expires using an optional refresh token or by running the authorization process again.
   - OAuth delegated authentication and access flow. The application asks the user if they want to give it access to resources on a third-party trusted service (resource server). If the user accepts, an authorization grant code is generated. This code can be exchanged for an access token with the authorization server. To make sure the authorization server knows the application, the application has to send an app ID and app secret along with the authorization grant code. The access token can then be used to access protected resources within a certain scope from the resource server.
   - Implementing an OAuth server on a Linux-based embedded device such as the Pi or the Intel Edison isn't hard because the protocol isn't really heavy. But maintaining the list of all applications,

users, and their access scope on each Thing is clearly not going to work and scale for the IoT.

(a) OAuth Roles

- A typical OAuth scenario involves four roles
  i. A resource owner—This is the user who wants to authorize an application to access one of their trusted accounts; for example, your Facebook account.
  ii. The resource server—Is the server providing access to the resources the user wants to share? In essence, this is a web API accepting OAuth tokens as credentials.
  iii. The authorization server—This is the OAuth server managing authorizations to

access the resources. It's a web server offering an OAuth API to authenticate and authorize users. In some cases, the resource server and the authorization server can be the same, such as in the case of Facebook.

i. The application—This is the web or mobile application that wants to access the resources of the user. To keep the trust chain, the application has to be known by the authorization server in advance and has to authenticate itself using a secret token, which is an API key known only by the authorization server and the application.

4.2.3 The Social Web of Things

- Using OAuth to manage access control to Things is tempting, but not if each Thing has to maintain its own list of users and application. This is where the gateway integration pattern can be used.

- use the notion of delegated authentication offered by OAuth, which allows you to use the accounts you already have with OAuth providers you trust, such as Facebook, Twitter, or LinkedIn.

- The Social Web of Things is usually what covers the sharing of access to devices via existing social network relationships.

1. A Social Web of Things authentication proxy

- The idea of the Social Web of Things is to create an authentication proxy that controls access to all Things it proxies by identifying users of client applications using trusted third-party services.
- Again, we have four actors: a Thing, a user using a client application, an authenti- cation proxy, and a social network (or any other service with an OAuth server). The client app can use the authentication proxy and the social network to access resources on the Thing. This concept can be implemented in three phases:
  (a) The first phase is the Thing proxy trust. The goal here is to ensure that the proxy can access resources on the Thing securely. If the Thing is protected by an API token (device token), it could be as simple as storing this token on the proxy. If the Thing is also an OAuth server, this step follows an OAuth authentication flow, as shown in figure 9.6. Regardless of the method used to authenticate, after this phase the auth proxy has a secret that lets it access the resources of the Thing.
  (b) The second phase is the delegated authentication step. Here, the user in the client app authenticates via an OAuth authorization server as in figure 9.6. The authentication proxy uses the access token returned by the authorization server to identify the user of the client app and checks to see if the user is authorized to access the Thing. If so, the proxy returns the access token or generates a new one to the client app.
  (c) The last phase is the proxied access step. Once the client app has a token, it can use it to access the resources of the Thing through the authentication proxy. If the token is valid, the authentication proxy will forward the request to the Thing using the secret (device token) it got in phase 1 and send the response back to the client app.
- All communication is encrypted using TLS
- Social Web of Things authentication proxy: the auth proxy first establishes a secret with the Thing over a secure channel. Then, a client app requests access to a resource via the auth proxy. It authenticates itself via an OAuth server (here Facebook) and gets back an access token. This token is then used to access resources on the Thing via the auth proxy. For instance, the /temp resource is requested by the client app and given access via the auth proxy

forwarding the request to the Thing and relaying the response to the client app.

2. Leveraging Social Networks

   - This is the very idea of the Social Web of Things: instead of creating abstract access control lists, we can reuse existing social structures as a basis for sharing our Things. Because social networks increasingly reflect our social relationships, we can reuse that knowledge to share access to our Things with friends via Facebook, or work colleagues via LinkedIn.

3. Implementing Access Control Lists

   - In essence, you need to create an access control list (ACL). There are various ways to implement ACLs, such as by storing them in the local database.

4. Proxying Resources of Things

   - Finally, you need to implement the actual proxying: once a request is deemed valid by the middleware, you need to contact the Thing that serves this resource and proxy the results back to the client.

4.2.4 Beyond book

- But just as HTTP might be too heavy for resource-limited devices, security pro- tocols such as TLS and their underlying cypher suites are too heavy for the most resource-constrained devices. This is why lighter-weight versions of TLS are being developed, such as DTLS,26 which is similar to TLS but runs on top of UDP instead of TCP and also has a smaller memory footprint

- device democracy.27 In this model, devices become more autonomous and favor peer-to-peer interactions over centralized cloud services. Security is ensured using a blockchain mechanism: similar to the way bitcoin transactions are validated by a number of independent parties in the bitcoin network, devices could all participate in making the IoT secure.

4.2.5 Summary

- You must cover four basic principles to secure IoT systems: encrypted commu- nication, server authentication, client authentication, and access control.
- Encrypted communication ensures attackers can't read the content of mes- sages. It uses encryption mechanisms based on symmetric or asymmetric keys.
- You should use TLS to encrypt messages on the web. TLS is based on asymmetric keys: a public key and a private server key.
- Server authentication ensures attackers can't pretend to be the server. On the web, this is achieved by using SSL (TLS) certificates. The delivery of these certif- icates is controlled through a chain of trust where only trusted parties called certificate authorities can deliver certificates to identify web servers.
- Instead of buying certificates from a trusted third party, you can create self- signed TLS certificates on a Raspberry Pi. The drawback is that web browsers will flag the communication as unsecure because they don't have the CA certifi- cate in their trust store.
- You can achieve client authentication using simple API tokens. Tokens should rotate on a regular basis and should be generated using crypto secure random algorithms so that their sequence can't be guessed.
- The OAuth protocol can be used to generate API tokens in a dynamic, standard, and secure manner and is supported by many embedded Linux devices such as the Raspberry Pi.
- The delegated authentication mechanism of OAuth relies on other OAuth pro- viders to authenticate users and create API tokens. As an example, a user of a Thing can be identified using Facebook via OAuth.
- You can implement access control for Things to reflect your social contacts by creating an authentication proxy using OAuth for clients' authentication and contacts from social networks.