IOT - P2P
Exam notes

Lukas Jørgensen, 201206057

March 28, 2016

Contents

1 General

1.1 Degrees of P2P Structure

- Unstructured networks
- Semi-structured networks
- Structured networks

Compare types of network

1.2 P2P Characteristics

- Scalability
- Performance
- Availability
- Fairness
- Integrity and authenticity
- Security
- Anonymity, deniability and censorship resistance

2 Unstructured P2P networks

2.1 Napster

Single point of failure in its lookup- / index-server.

2.2 Gnutella

- The first major truly distributed P2P network.
- Quite primitive, yet hugely successful

Commands

Ping	Ask for information about a peer.
Pong	The response to a Ping.
Query	Ask for the address for some information.
QueryHit	The response to a query.

`QueryHits` follow the query-route back to its originator.

Queries have a unique ID (UID), and a time-to-live (TTL) to make sure that the same query is not re-transmitted and that queries die out.

Drawbacks

Peers report:

- Amount of shared data.
- Available bandwidth.

Self reporting lead to free-loaders.

Flooding generates a lot of duplicate network traffic, Gnutella can swamp the network, even without any data-traffic. You could use Ring instead, but it has a worse worst-case.

Downloads the whole file from a single peer, so if that peer dies, so does your file-transfer.

2.3 k-Walkers

A walker: A query traverses the network randomly, until a match is found. k-Walkers is simply having k queries traversing.

Much more efficient in terms of traffic, user-perceived delay is longer.

Variations: $Maintainstate$, and do not forward the same query to the same neighbour twice. $Check$ source for sufficient success. $Check$ visits fewer nodes than the other variation.

Results

Scales much better than flooding. Increased user-perceived delays. Blindly using TTL is inefficient. Queries should check back.

However simulation assumes a stable network, and it may not be distributed in a nice way.

2.4 Gia

A system combining:

- **Topology adaptation** - peers should connect to strong and well-connected peers able to handle the traffic.
- **Active flow control** - if a peer is overloaded it should not be bothered until it is ready again.
- **One-hop replication** of indices - every peer knows what its neighbours store.
- **Biased random walking** - queries seek towards high capacity peers.

Terms

Capacity ability to handle messages/time - i.e. bandwidth, CPU power etc.

Satisfaction $0 \ldots 1$: degree to which a peer's own capacity is matched by the sum of its neighbours' capacities/degree.

Adaptive flow control

Peers sends tokens to its neighbours according to its (and their) capacity. Peers only sends queries to other peers if they have a token from that peer. So if they claim to have a low capacity, the other peers will send it fewer tokens, which they need in order to get work done. To reduce overload, publish fewer tokens and queue queries.

One-hop replication

Peers maintain indices over their neighbours' resources. Query results will contain a pointer to the resource, not the index. This evens the load for peers with many resources.

Biased random walking

Walkers will walk towards the strongest peers in the hub. I.e. peers will direct it to the node with the highest capacity tokens.

3 Structured P2P networks

3.1 Distributed hash tables

Assign peers evenly across an ID-space.

Assign resource IDs in the same ID-space. Resources are associated with the following (in ID-space) peer.

Searching for a resource and for a peer becomes the same thing.

3.2 Chord

One operation: `address = lookup(key);` Given a key, find a node responsible for that key.

Goals

- Load balancing
- Decentralization
- Scalability
- Availability

- Flexible naming
- Lookup in $\mathcal{O}(\log(n))$
- Each node needs information about $\mathcal{O}(\log(n))$ peers

Hashing in Chord

A good hash function balances load (uniformly distributes them). Even small changes, results in big differences in hashes. Cryptographic hashes, such as SHA-256, is very hard to get the key from the hash.

Big enough space to make collisions improbable.

Node leaving

All of node n's keys are assigned to `successor(n)`.

Node joining

Keys $k \leq n$ assigned to `successor(n)]` are assigned to n. A physical node may run any number of peers on different ports.

Key location

Linear: takes $\mathcal{O}(n)$ time, and $\mathcal{O}(1)$ space.
Scalable: use fingertables, point to the 2^{i-1} next node. n.finger[i] = find_successor($n+2^{i-1}), 1 \leq i \leq m$.

Simply find the node n' in the fingertable, which is closest to the key. This node will be the one in the fingertable, that will know most about n'.

Successor lists

If a node fails, use successor list of size s to re-establish the network.

Conclusion

- Based on distributed, consistent hashing.
- Performance and space is $\mathcal{O}(\log(n))$ for stable networks.
- Simple, provable performance and correctness.
- Too simple, does not consider locality or strength.

3.3 Pastry

- Effective: $\mathcal{O}(\log(n))$ routing hops.

- Distributed: no servers, and limited knowledge at nodes ($\mathcal{O}(\log(n))$ routing table sizes).
- Substrate: Not an application, rather an application programming interface (API) to be used by applications.
- NodeId: similar to Chord.

API

- `nodeID = pastryInit(Credentials, Application);`
- `route(msg, key);` routes a message to the live node, with its ID numerically closest to the key.
- (Callback, application implements) `deliver(msg, key);` called on the application at the destination node for the given id.
- (Callback, application implements) `forward(msg, key, nextId);` called on the application, when the node is about to forward the given message to the node with $nextId$ as id.

Assumptions and guarantees

Routes messages in $\log_{2^b}(n)$ steps. Where b is the base the id's are in.

Unless $|L|/2$ adjacent nodes fail concurrently, eventual delivery is guaranteed. ($|L|$ being a configuration parameter).

Join and leave in $\mathcal{O}(\log(n))$.

Maintains locality, based on application-defined scalar proximity metric.

Routing table

3 parts, `Leaf Set`, `Routing Table` and `Neighbourhood Set`.

Leaf Set

Contains the $|L|$ numerically closest smaller nodeIds, as well as the $|L|$ numerically closest larger nodeIds.

Routing Table

Has `ceiling`$(\log_{2^b}(N))$ rows with $2^b - 1$ entries.

Each entry in row n shares a prefix of length n with the current node.

Neighbourhood Set

$|M|$ closest nodes according to proximity metric. (M begin a configuration parameter).

Routing in Pastry

The node first checks if the key is within the range of the `Leaf Set`. If not, it uses the routing table to forward the message to a node that shares a common prefix with the key by at least one digit.

In the rare case that the appropriate entry is empty or unreachable, then the message will be forwarded to some known node, with a prefix at least as good as the current node, and is numerically closer.

Node joining

A new node, n has to know about some nearby node m. n asks m to route a "join" message with key equal to n. `Pastry` then routes this message to node p with the nodeId numerically closest to n. All nodes on the route to p returns their state to n.

n updates its state, based on returned state:

- `Neighbourhood Set` = `Neighbourhood Set` of m.
- `Leaf Set` is based on `Leaf Set` of p, since p is numerically closest to n.
- Rows of the `Routing Table` are initialised based on the nodes visited on the route to p, since they share increasing common prefixes with n.

n then calibrates its `Routing Table` and `Neighbourhood Set` based on data from the nodes referenced therein.

n sends its state to all the nodes mentioned in its overall routing table. In total, $\mathcal{O}(\log_{2^b}(n))$ messages exchanged. (Since that's amount of nodes visited in the join message).

Locality

Applications are responsible for providing proximity metrics. Pastry assumes the triangle inequality holds.

Node failure

Leaf Set

- Contact the live node with the largest index on the side of the failed node, and get leaf set from that node.
- Returned leaf set will contain an appropriate node to insert.
- This works, unless $|L|$ adjacent nodes have failed.

Routing Table

- Contact other node on the same row, to check if this node has a replacement node (the contacted node may have a replacement node on the same row of its routing table).
- If not, contact node on next row of routing table.

Neighbourhood Set

- Neighbourhood set is normally not used in routing, therefore contact periodically to check liveness.
- If a neighbour is not responding, check with live neighbours for other close nodes.

3.4 Kademlia

Routing done by halving the ID-space distance in each routing step. Similar to the `Routing Table` in Pastry's routing table.

Routing done in $\mathcal{O}(\log(n))$, space used is $\mathcal{O}(\log(n))$.

Critique of other systems

Chord:

- Fingertables only forward looking
- Messages arriving at peer tells it nothing useful, knowledge must be gained explicitly
- Seperate track of control message exchanges
- Rigid routing structure
- Locality difficult to establish

Pastry:

- Complex routing algorithm
- First routing table, then leaf set
- Maintains three different tables: leaf, routing and neighbour

IDs

All IDs are 160 bits long, found using a uniform distribution function, such as SHA-256. Kademlia uses xor to navigate this ID space. $d(X, Y) = X \text{ XOR } Y \implies d(X, Y) = d(Y, X)$

Intuition: The more significant the different bits are, the longer distance there are between them.

Routing table

A Kademlia routing table, stores 160 k-buckets. The i^{th} k-bucket, contains nodes within a `XOR` distance of 2^i to 2^{i+1} from itself (so the i^{th} bit is significant).

There can be up to k nodes in each bucket, ordered by liveness (most recently seen are positioned at tail). So they have finest grained knowledge about the closest peers, and coarser knowledge about the rest of the world.

Commands

Ping check if a peer is still alive.

Store store some key-value pair at the peer.

Find_Node Returns the k closest nodes to an ID, that the peer knows.

Find_Value Similar to `Find_Node` but if the node knows the value, it returns the value instead. If one of the k closest nodes does not have the value, the requester will store it there.

Maintenance of routing tables

Maintenance is ongoing, when communication with another node happens:

- Check the appropriate k-bucket.
  - If already there, move to tail.
  - If there is room, insert at tail.
  - If unknown, and least recently seen node is unresponsive (at the head), replace with new node (and move to tail).
  - Otherwise, do nothing.

Over time, this means that old peers will stay in the buckets, as long as they are alive. This indirectly favors old peers that have stayed in the buckets for a long time (e.g. k old peers in one bucket, will never be replaced by a new peer if they stay alive). The longer a peer stays online, the higher the probability is that it will remain online.

Protects from flooding with bogus peers.

Parallelism

In Kademlia, we can send queries to multiple nodes, and just use the response from the quickest node. This prevents one peer from slowing bottle-necking the network.

Redundancy

Data expires after 24 hours, so original publisher has to republish every 24 hours to maintain information. Each (key,value) pair is stored at k locations close to the key.

Whenever a peer A observes a new peer B with ID closer to some of A's keys, A will replicate these keys to B.

Joining the network

- Compute an ID
- Somehow locate a peer in the network
- Add the peer to the k-bucket
- Find neighbours with `Find_Node` on own ID
- Populate the other k-buckets by performing `Find_Node`
- Since we issue queries on other nodes, those nodes will implicitly be added to the appropriate k-buckets for these nodes.

Failure

Unlikely, as routing tables are maintained by ordinary traffic. If there is no traffic, a peer will regularly refresh oldest k-bucket to keep updated.

Parallelism in queries will ensure that a failing peer is both detected and bypassed.

4 The Internet of Things

4.1 What is the IoT

The internet of things (IoT) is 3 main parts:

- Identity - RFID, QR, MAC-address
- Connectivity - How can we address the object? Bluetooth, WiFi, IR etc.
- Capacity
  - Simple: Identification
  - Intermediate: Sensing
  - Advanced: Reacting

4.2 What can we use it for

Logistics, intruders, smart homes etc.

4.3 Challenges

- Energy usage
- Lack of IP Addresses. 32 bit not enough anymore, IPv6 = 128 bit
- Lack of standards, how are devices found, named, how are their abilites discovered etc.
- All major players are running with their own solution, no one is big enough to be the standard.

Data silos

- Pre-Web internet: Data flowed evenly between hosts.
- Present internet: Data flows from content-providers to end-users; data about habits collected.
- Future IoT internet: Countless sensors and devices, streaming data to central repositories (clouds).

Being the owner of the standard cloud service for IoT is a major opportunity for a lot of big players, as they get access to all that data.

Security

A "smart" home, is a hackable home.

4.4 The Web of Things

A proposal for a solution, use the already existing platform, the world wide web (WWW), and use the existing protocols.

- GET
- PUT
- POST
- DELETE

4.5 Disadvantages

- HTTP is heavy.
- HTTP was not designed for streams.
- Security and discovery are both issues we can handle, but has to be handled explicitly.

5 Security and privacy in P2P and the Internet of Things

5.1 Dangers of distributed systems

- Trust - Who can you trust?
- Identity theft - Pretending to be you or someone you trust
- Privacy - Preventing others from listening in on the conversation
- Censorship and attacks - Denying you the right to know

Data travels through numerous computers before it gets to the destination. This means data can always be intercepted, so we need make sure the interceptor cannot read it, as it will be useless to him.

5.2 Attacks agains distributed systems

DDoS

Simply overload the system, often with a botnet.

As distributed denial of service (DDoS) attacks usually focus on overloading one peer, and not all the peers at once. We should focus on how to protect single peers, and protect the network from losing a single peer.

Defences:

- Minimize cost of losing *any* peers.
- Make it difficult to identify the most important peers, so the attackers will have a hard time targetting them.
- Optimise traffic so it only affects a minimal part of the network.
- Prevent bogus data from overwriting good data.

Malicious peers

Peers that reroutes data in the wrong way, poisons routing data for other peers, corrupts data etc.

Defences:

- Do not rely on just one query route.
- Continuously very peers and data.
- Favour long living peers.

Sybil attack

Creating **a lot** of fake peers and making them join the network, easy to do if one machine can masquerade as many.

Allows surveillance or manipulation of traffic.

Defences:

- Make joining expensive, so it is harder to do from a single machine.
- Make sure that paths on the overlay network involve multiple subnets, as sybil attacks are likely to originate from the same subnet, so involving multiple subnets can mask who the sender and receiver is.

Eclipse attacks

Good peers are eclipsed by evil peers, that insert themselves between good peers and the network, cutting the good peers off from the network.

Defences:

- Prevent peers from choosing their position in the network.
- Don't rely on a single path through the network.

5.3 Techniques for anonymity and censorship resistance

Crowds

This aims to defeat network tracking.
Redirect requests through a number of random peers, before doing the request.
Obscure who sent the request.

Mix networks

This aims to defeat traffic analysis.
Send messages through a number of mixers $m_1 \dots m_n$, and encrypt the message with m_n's public key (PK), then m_{n-1} PK down to m_1. Only m_1 knows the sender, and only m_n knows the receiver and neither knows the route of the message nor their position on the path.

Issues:

- Easy to block established mixers.
- Malicious mixers can recognize sender and recipient at that point.
- Doesn't protect the edges of the cloud (entering and leaving the mix network).
- It becomes feasible, but expensive to do edge traffic analysis.

Tarzan

An attempt to fix the issues with existing mix networks.

- It's P2P - everyone can be a mixer, therefore hard to block all the mixers.
- Fake "cover" traffic, makes it hard to distinguish real traffic from the idle chattering.
- Cover traffic is sent at a uniform rate, and lowered when real traffic comes, so there wont be an increase in traffic.
- Peers must be validated, and you cannot fake your IP return address.
- Tarzan spreads traffic across the internet, so malicious peers can't make a lot of virtual peers.

Freenet

A distributed filesystem, peers contribute diskspace and only a file-owner can alter a file.

There is end-to-end encryption, and only the originator of a query knows it is the originator. Query results (files) are returned along the same route, during which peers might cache the file and claim that they are the data-holders. Thus hiding the actual data-holder from attacks. Peers might also alter the TTL, to avoid TTL analysis.

Popular resources are then replicated over the network, making DDoS attacks self-defeating as the more you attack a resource, the more that resource is spread across the network.

Joining

Create a globally unique ID (GUID) by hashing the resource, and send it out on the network with a TTL. Unpopular files might be reclaimed by the system, to make room for popular files.

5.4 Securing DHT's

Vulnerabilities

- Deterministic routing
- Routing information kept at peers
- ID determines position
- Values kept at peer with closest key

Kademlia

Weaknesses

- Deterministic routing along converging path
- Sybils can saturate the network with malicious peers
- Eclipse peers can work together to create poor routing

Strengths

- Prefers long living peers, so churn attacks will be inefficient
- Routing information is continually refreshed, there is no specific operation that can be targeted

Securing Kademlia

Everybody has public/secret keys
Securing Kademlia through:

- Expensive nodeId generation
- Sibling broadcast
- Ensuring that when we search for ID's we search across disjunct paths
- Verifiable messages

Secure Node Identifiers

Sybils rely on cheap, home-made, unverifiable nodeId generation.

- ID's created as public-key hashes
- Weak signatures on just (IP,port,timestamp) for `PING, FIND_NODE` commands. The timestamp will avoid replay attacks.
- Strong signatures on the whole message to avoid man-in-the-middle attacks, also include a nonce to avoid replays.

Generating ID's: If we have a central authority, it can control the ID-generation and limit the growth of Sybil's. But having a single point of failure (SPoF) is a bad idea, as it is contrary to the whole point of our peer to peer (P2P) network.

Alternative: Security by difficulty, use crypto-puzzles to make it computationally hard to create a nodeId.

This could be that for the generated key: c_1 first bits of $H(H(key)) = 0$ which means we have to generate a lot of keys before we get our public key. Then it becomes computationally infeasible to generate a nodeId with a hash that gets placed between two specific nodes.

To make it even harder, we add a c_2 and demand that the peer creates a X such that c_2 first bits of $H(key \oplus X) = 0$, then we can increase c_2 over time to keep nodeId generation expensive.

Verification is $\mathcal{O}(1)$ while creation is $\mathcal{O}(2^{c_1} + 2^{c_2})$.

Ordinary peers just have to solve the puzzle once, but in Sybil attacks, you have to solve it thousands of times which is very expensive.

Sibling broadcast

Standard Kademlia has k buckets and k replicas of key/values *(siblings)*. This marries network connectivity to redundancy.

In the secure Kademlia, we instead have s replicas of key/values. Unclear

Populating the k-buckets

If you are able to provide a signed response, and if there is room in k-bucket, you can be added.

Only adds valid (signed) nodeIds if the prefix is sufficiently different. Unclear

Querying in S/Kademlia

Instead of just using the peer that answered quickest, it divides the k-peers into d-groups. Each of these groups, searches for the query as usual, with the exception that, if a group has visited a peer, the other groups will not visit this peer. This means their paths are disjoint.

This will reduce the likelihood that the whole query gets poisoned. It's important to note that it is not impossible for the whole query to get poisoned, but it is just a lot more expensive than ordinary Kademlia.

Conclusion

Making attacks harder by:

- Making nodeId generation harder with Crypto-puzzles.
- Accepting only signed nodeId's into k-buckets.
- Distributing queries across a wider set of the network.

At the cost of having good peers solve crypto-puzzles. (Joining takes longer).

5.5 Securing the Internet of Things

A lot of devices, involving a lot of different systems, architectures solutions and actors. What could possibly go wrong?

What if someone hacks our water-supply, or the lock to my door?

Types of attacks

- DDoS - annoying if I can't use my toaster, catastrophic if the power-grid goes down.
- Surveillance - by companies, criminals or even the state.
- Intrusion - if one device gets hacked, it could potentially compromise all devices within the network. E.g. circumventing a firewall

Network level security

Different types of devices:

- Strong cryptography simple to implement on ordinary computers, what about smaller simpler devices?
- Public key infrastructure may be difficult to handle in a larg IoT setting.
- Gateways is a possible solution, the gateways handle the security and the things can keep doing what they do.

A centralistic solution is simple, but that is a SPoF and if that is hacked then it will grant access to all the devices. Unclear

Privacy

- Unless we can ensure that the users data belongs to the user, then IoT could become the perfect surveillance infrastructure.
- Gathering the users data at one point on the internet (centralistic) makes it easier to exploit.
- Distributed solutions keeps data closer to the user, so all the data isn't gathered in one place, and forces attacks to be targeted personally at a user.

Identity

- IoT objects must have identities that can be found and authenticated by other services.
- Easy to implement in a centralised system, but in a distributed system, having a coherent namespace is challenging, as devices join and leave.

Trust

- Between the user and the system
- Also between devices, how does devices know that they can trust each other?

Fault tolerance

- Systems must cope with failures, things will go wrong with high amount of devices.
- Identify errors and failing sensors. Automatically choose alternate sensors or services.
- Identify compromised systems, and route around them if they are under attack.

5.6 Conclusion

With the IoT we get all the security and legal issues possible. Both technical issues, protection from hacking and attacks as well as the question of who owns the data? The provider of the service, the provider of the device etc..

Could add more here, about security benefits between IOT and central

6 Mobile ad hoc networks

Ad-hoc networks are small local networks for devices to communicate with eachother, as opposed to connecting to the "standard" network, such as the internet.

Useful if, for some reason, one doesn't want to use the standard network. It might be expensive, slow or not available (or for any other reason).

6.1 Radios

Radios is by far the most common wireless communication technique. It has some issues:

- Signal strength diminishes with distance, and may vary a lot and seemingly unpredictably.
- Can be unidirectional
- Signals can be blocked by environmental factors, like walls, mountains or even rain.

- Signals are always broadcast, can be security problem but can also be an optimization.

6.2 Routing

The issue with finding the "cheapest" path between devices. There might be a delay or cost for sending a packet from A to B, that is smaller if we send it from A to C to B. This could be because of environmental factors.

Wired networks

Implementation through actual wires.

- Nearly static configuration.
  - Nodes stay in the same place for a long time.
  - Interconnecting links exist for an extended period of time.
- Small fluctuations in link quality - usually only fluctuates when very high traffic rates are experienced.
- High traffic rates - there's usually a lot more communication going on in these types of systems.

Classical routing approaches

- Link State protocols
- Distance Vector protocols

Both build full routing tables.

Link State

It is assumed that the network topology and all link costs are known.

This means that global information is needed and must be communicated throughout the network periodically - a lot of traffic.

For each node in the table, the routing table contains an entry with the distance to the node and the ID of the next neighbour in that path. These distances can be computed with single-source-shortest-path (Djikstra).

Periodically, and when there are changes in link costs, nodes broadcast information about their outgoing links to the entire network.

This scales poorly, and is best for relatively small, stable, networks.

Upon changes, the entire routing table is recomputed. This is an $\mathcal{O}(m + n\log(n))$ operation, where n is number of nodes, and m is number of edges (wires).

Distance Vector

In Link State, a node A would have to build information about the whole network. In Distance Vector, we only exchange information between the immediate neighbours.

When an update occurs, only the affected parts of the routing table is updated, and the changes are propagated to neighbour nodes.

We use the Bellman-Ford algorithm for this.

Properties:

- It quickly enters a resting state just like Link State.
- But it also works well in unstable networks:
  - Routing tables are updated incrementally.
  - Updates are only send to neighbours and only propagate if a new shortest path is found.

Problems: Count-to-infinity, if A -> B -> C -> D has length one at each link, and the link from C to D is broken, then C thinks that B has a route to D, and B thinks that C has a route to D and so on. They will keep incrementing the distance to D, and will converge towards infinity.

Summary

- Routing is: finding the cheapest path between two nodes in a graph. (According to any metric)
- Most classical routing protocols are based on either Link State or Distance Vector protocols.
- Link State is most efficient in small stable networks.
- Distance Vector builds its routing table incrementally and is thus more suited in larger, unstable networks.

6.3 MANET routing

Desirable properties

- Minimal control overhead.
- Minimal processing overhead - we usually deal with small devices that can't handle large computations.
- Multi-hop routing capability - not much of a routing system if we can't calculate multi-hop routing.
- Dynamic topology maintenance
- Loop prevention

Proactive routing

- Nodes periodically exchange routing information and attempt to maintain routing information of the entire network.
- Ensures efficient routing, at the cost of constantly exchanging routing information between peers.
- Relatively high control overhead in low traffic scenarios.
- Using the "fresh" routing table, is a big advantage in high-traffic scenarios.
- Good for low-latency requirements.
- Very expensive for dynamic networks.
- You might keep routing information for scenarios that are never relevant.

Reactive routing

- Route planning is done on-demand. Nodes only try to find a route to a destination when actually needed.
- No constant maintenance overhead, nodes do not maintain state.
- May save a lot of control overhead, as it is only sent when needed.
- Routing is costly and unpredictable.
- Initial latency for a route is high - subsequent latency is lower.
- Very effective in low traffic scenarios.

Destination Sequenced Distance Vector

A pro-active approach.

A Distance Vector algorithm that:

- Uses sequence numbers to avoid routing loops.
- Is more bandwidth efficient through the use of incremental updates.
- Delays route advertisements to reduce fluctuations.

Routing table

Each node maintains a table of:

- Destination
- Next Hop
- Metric (associated cost0)

- Sequence Number (how new is the information - higher is newer)

Whenever it reacts to updates, the route with the highest Sequence Number is always preferred.

Whenever a route update is sent out, it increments the sequence number with two. Therefore, any network information originating from the node itself will always be equal numbers.

If a link breaks, the node who discovers the break sends out an update with the cost set to ∞ and increments the sequence number by one.

Any subsequent update sent by the disappeared and reappeared node will automatically supersede this message. (Since it the Sequence Number with two so it will be one higher than the ∞ message).

Updates in DSDV are bundled together to bring down the overhead of sending this data.

- Full updates are sent infrequently - includes the full routing table, used to bootstrap new neighbours.
- Incremental updates are sent frequently - Only information about changed routes, must fit within a single packet, otherwise full update is sent.

Ad-Hoc On-Demand Distance Vector

A re-active approach.

Uses destination sequence numbers to avoid loops just like DSDV. Routes are discovered by broadcasting route requests (RREQs) containing:

- Source address
- Source sequence number
- Broadcast ID
- Destination address
- Destination sequence number
- Hop count - the number of hops the request made.

When the RREQ reaches a node that has a recent route to the destination, a route reply (RREP) is sent back to the source using the reverse path.

There could be multiple RREPs in this case we prefer the one with the highest sequence number (newest) or, if they are equal, the one with the lowest hop-count.

As the RREP is sent back through the network, the intermediate nodes record it who they received it from, and thus a forward path is built.

Upon link failure: if a link fails, the upstream neighbours sends a RREP with Sequence Number incremented by one, and Hop Count set to ∞ to any active neighbours, i.e. neighbours using that route.

Rediscovery: Again, like in DSDV it simply uses a higher sequence number and thus overwrites the RREP sent out upon link failure.

Properties of AODV

- Minimises control traffic, as it only maintains active routes.
- Processing overhead is very small, as it only does simple table lookups.
- Functions well in low-traffic scenarios.
- In high-traffic scenarios, it can have a very high initial cost - especially in unstable networks, as routes must be recomputed often.

Dynamic Source Routing

A re-active routing protocol using a discovery procedure similar to the one used in AODV.

RREQ and RREP include the entire routing path, so intermediate nodes do not have to keep any any state while forwarding RREPs and RREQs.

Does *not* require bi-directional links.

Instead of using the reverse path, a node can send a RREP by either using an already known path to the source, or sending out a RREQ to discover a path to the source.

Preventing looping is simple, just make sure there are no loops in the routing path, as we have the whole path.

The routing table

Instead of a conventional routing table, the node maintains a cache routes to destinations, which is in effect a tree with root in the node.

The cache may contain multiple routes to the same destination, this can be used for graceful handling of route errors, and just send a RREP to the source with the alternative path.

Promiscuous mode operation

Radio is a broadcasting medium, so we can eavesdrop on messages. This can be used in DSR:

- Reacting to error messages, and proactively mend route cache.
- Reacting to messages with itself as the destination, and tell the node that the intermediate node can be skipped.

Summary

- mobile ad hoc networks (MANET) are mobile ad-hoc networks
  - High mobility leads to the need for new routing approaches
  - Lack of infrastructure means that participating nodes must route themselves
- MANET routing protocols fall into one of two categories:

– Proactive algorithms tries to keep an up-to-date routing table at all times.
– Reactive algorithms build routing tables only when needed.

6.4 Energy-efficient MANET routing

Why energy efficiency?

Nodes in a MANET, are mobile and therefore usually battery powered, and network interfaces consumes a lot of energy.

Nodes route each others data, which means they are continuously helping each other, which costs battery-time. All nodes are needed to avoid fragmenting the network, so if crucial nodes go down it could hurt the network.

Some sensors may be discarded when they run out of battery, thus increasing the lifetime, increases the utility of the sensor.

The costs for one particular wi-fi device have shown:

- Transmitting costs: 1.4 W
- Receiving costs: 1.0 W
- Idling costs: 0.83 W
- Sleeping costs 0.13 W

Broadcast media spends a lot of time receiving useless packets as they were not meant for them.

Power control

Power-control schemes are based on the fact that there is a non-linear relation between transmission range and energy used to transmit. Sometimes taking two short steps instead of one long step saves power.

Power-control schemes use less power by using a shorter transmission range. Which gives us:

- Lower total energy consumption for sending a packet.
- Less noise, as only close neighbours are disturbed when a message is sent.

Power save

Based on the fact that, in most MANETs idle time is dominant. So most of their energy is spent while doing nothing. So we can utilise the sleep states of the wireless interface to put the nodes to sleep.

Challenges

How do we maintain connectivity when some of the nodes are sleeping?

- By using retransmissions
- By making sure that enough nodes are kept awake at all times
- Or a combination of the two

BECA

Simple power-save approach based on retransmissions. Retransmit for so long that we know the node will be awake to receive at some point. Listen interval: $T_l = T_0$, retransmission interval: T_0. Sleep interval: $T_s = kT_0$

Could save $\frac{k}{k+1}$ power save, at the expense of large discovery latencies. When nodes on the path to a destination node are sleeping, we have to wait for them to wake up, this gives us a worst case delay of kT_0 **for every routing hop**. On the other hand, if you can discover the network in time for the network to still be awake once it has been discovered, then there will be no further delays.

AFECA

Extends BECA by taking node density into consideration.

If the node density is high, there is a high probability that some node in the network will be awake to perform routing tasks. Therefore individual nodes can sleep for longer, making the new sleep interval:

$$T_{sa} = Random(1, N) \times T_s$$

6.5 Wireless Sensor Networks

A network that consists of a a lot of nodes, that are dispersed in some environment in order to measure some phenomenon (moisture, forest fires, hurricanes etc.).

We want to be able to take a whole bunch of peers, and just distribute them in the environment. Then they should be able to establish a MANET network.

It's a specific case, since we have a lot of sensors which collect data, and want to send the data to some data sinks. So all the data from the sensors has to go in the same direction.

The nodes are cheap, perhaps even disposable. The most important requirement is the survival of the network, not the individual nodes.

Energi-concerned routing

What is the goal, of the peer etc..

- Lowest energy-usage route
- Fewest hops route (lowest latency)

- The route that travels along the peers with most battery (avoid low-battery peers to avoid draining them)
-

Data-aggregation

Since all data moves from sensors, towards the sink. Therefore we can aggregate the data, which is much more efficient. Depending on the scenario, we might even be able to do processing along the way.

Data-centric routing

In ordinary MANET, we might request a resource held by a specific node.

In WSN, queries are data centered.

- Sinks can request data matching certain attributes.
- Nodes can advertise that they have data (meta-data often cheaper than actual data).

7 BitTorrent

- Tracker - A centralised peer-discovery entity
- Seeder - Peers that have fully downloaded the file being shared
- Leecher - Peers that are actively downloading the file
- Swarm - All the peers participating in sharing the data, most peers only deal with a subset of the swarm - their personal *peer set*
- .torrent file - a meta data file containing information about the torrent. Contains information about the files being shared as well as the trackers

7.1 The protocol

You get the .torrent file somehow, e.g. from a website or an email. In the .torrent file, you can extract the "announce" field, which is the tracker, and contact it to get a list of (50) peers.

After this point in time, the tracker is only contacted when leaving the swarm, every 30 minutes to show that the peer is active and if the peer is running low on peers in its set.

After retrieving the list of peers, the new peer establishes a connection to about 30 of these peers, thus the peer enters into a neighbourhood of peers and starts following the *peer protocol*.

7.2 The peer protocol

Entering a new neighbourhood

The peer sends a bitfield message, which as list of 1-bit boolean telling if it got the *ith* piece.

Downloading

When a peer is interested in downloading a piece, it sends a `piece` message with the index of the piece.

Each peer in the neighbourhood list has two state bits:

- interested/uninterested - Is the neighbour interested in the pieces we've got
- choked/unchoked - Are we allowing peers to download pieces at this point in time?

Peers send `choke/unchoke` and `interested/uninterested` messages to eachother in the peer protocol.

Choking

If we are downloading from a peer, we will unchoke it so it may also download from us. So we should prefer peers that are also interested in us. If we are not able to download from a peer, we can choke it again.

Optimistic unchoke: One or more peers will be unchoked every 30 seconds, if it then starts contributing, it will stay unchoked.

Choked/unchoked state is reconsidered every 10 seconds. At any point in time, peers should have some amount of unchoked neighbours (implementation specific).

Seeding

When seeding, we tit-for-tat doesn't make sense. A seeder works for the general good of the swarm, and therefore aims to upload as much as possible as fast as possible. Therefore, it unchokes peers to which it has a high upload rate.

Piece-selection

Use some strategy, could be *random* or *rarest-first* to decrease the likelihood of the torrent "breaking" (the swarm lacks some pieces) when a peer leaves.

7.3 Attacks on Bittorrent

Can take on two forms:

Harming the swarm

Piece lying

A Sybil attack, where a lot of malicious peers join, and all claim that they have the rarest pieces, and `choke` peers trying to download from them. Then the *rarest-first* strategy will fail since they think the rare pieces are common. Then when the last seeder leaves, the torrent could break.

Eclipsing correct peers

This is done by adding a lot of malicious peers to the swarm. When a correct peer tries to download from one of the malicious peers, it will notify all the malicious peers and then they will all try to connect to the correct peer.

Taking advantage of the swarm

What if the attacker wants to take advantage of the swarm, rather than just destroying it?

A sybil attack

Strong peers seem to be uploading a whole lot more than downloading. They upload too much compared to how much they get in return.

A high capacity peer can disguise itself as a lot of smaller peers. This will increase the likelihood of tit-for-tat reciprocation so they can get more compared to how much they give.

Alternately, creating a lot of peers increases the likelihood of receiving optimistic unchokes, therefore being able to download without uploading.

Can be mitigated by disallowing connections from the same IP.

Adaptively resizing the active set

Active set is the peers we are currently uploading to and downloading from. Simply connecting to more peers at once. Standard implementations use a $\sqrt{(\text{Upload capacity})}$ but if you go for a linear amount of connections instead, you should be able to get a more even split.

BitTyrant's unchoke algorithm

For each of the neighbours p, BitTyrant maintains estimates for the upload rate u_p and download rate d_p.

Peers are then ordered by $\frac{d_p}{u_p}$ and unchoked in order until the sum of u_p's exceed the peer's upload capacity.

Furthermore, we start by giving them some upload rate, and then gearing it down until we the peer chokes us, then increase it a bit so we get unchoked and keep going for the minimum upload rate without getting choked.

- Performs well in a regular system, and BitTyrant systems where there are altruism.

- If no peer are altruistic, i.e. contribute excess capacity, then there is a performance takes a serious hit.
- It's a great optimization for high-bandwidth peers.

Conclusion

Adaptively resizing the active set is an OK optimization, but the unchoke algorithm is selfish.

8 Your own P2P system