Reliable Routing Protocols in Networks with Byzantine Failures

Matthew Kerner

March 10, 2007

1 Introduction

The network layer in a computer network is responsible for providing a datagram service, in which a packet issued from a source node to a destination node is expected to reach the destination node in a reasonably short period of time with some probability. The datagram service does not provide delivery *guarantees*: these are expected to be provided by a higher level transport layer, on top of a basic datagram service. But in order to design a reasonably efficient transport layer on top of a datagram service, the datagram service should provide the advertised service with high probability.

The network layer functionality is typically provided by routers connected by links. These routers implement a protocol, which can be considered to be a distributed algorithm: each router performs the algorithm locally, operating on local data stored in local data structures, and the overall result is a global pattern of behavior.

Many widely used routing algorithms are robust in the presence of simple failures (e.g. a router crashes and stops working completely, or a link goes down). However, a router with a Byzantine failure may act arbitrarily or even maliciously. We outline two network layer protocols that can route successfully in the presence of Byzantine failures as described in [2].

2 Problem

2.1 The Network Environment

First, we define the environment in which failures may occur. Consider a network consisting of a set of nodes, connected by point-to-point bidirectional links. We consider links to be "dumb" in that they are simply communication channels without any intelligent capabilities. Nodes are "smart" and can make decisions about how to behave. Data is transmitted between nodes in packets sent by sources, addressed to destinations. It may be costly to provide links between every pair of nodes: in such cases the data is expected to be forwarded by intermediary nodes from the source to the destination. Nodes with this forwarding capability are called routers (we use the two terms interchangeably below). The network layer protocol specifies the format of packets, the nature of addresses, and the expected behavior of routers when they receive packets of different kinds, with different data in them.

2.2 Network Layer Protocol Overview

Before we turn to the nature of failures, we also give a brief outline of how network layer protocols achieve data transfer. The most simple (and inefficient) communication algorithm in the network layer is flooding: a source sends a packet to a destination, and this packet is disseminated throughout the entire network by every node across every link (perhaps multiple times) to ensure that the packet reaches the destination. Flooding schemes usually implement a per-packet hop count or time-to-live that is decremented every time the packet is forwarded to prevent packets from circulating throughout the network indefinitely. In flooding schemes, if there is at least one functioning path between the source and the destination node with length less than or equal to the packet's time-to-live, the packet can travel from the source to the destination.

While flooding may be simple to describe and implement, it is not efficient. Packets that are flooded may travel across every link in the network multiple times as they make their way from the source to the destination when only a few (or even one) link traversal is really necessary. Furthermore, flooding can result in exponentially many copies of a packet being generated within the network as each node sends out duplicate copies of the packet on every incident link.

Because flooding is inefficient, most network layer protocols specify a more sophisticated and efficient mode of operation. One common approach is to have nodes keep track of the addresses of their neighbors. Neighbors can be detected by advertising ones presence across the links incident to oneself, and listening for such advertisements from neighbors over adjacent links (the packets used to do this are limited to travelling one hop, so only nodes direclty adjacent to the sender will receive them).

In a class of routing algorithms that are known as link state algorithms, per-node neighbor lists are then flooded throughout the network, this time to every other node in the network. Once a node receives such link state information from some other nodes in the network, it can compute routes between source/destination pairs using using one of many graph-based route-construction algorithms. Such paths can be used to route data efficiently between a pair of nodes. In some cases these routes are computed by the source, and in other cases these routes are computed incrementally and independently by each node along the packet's path.

2.3 Simple Failures

Most network-layer protocols are designed to handle simple failures of routers or links. For example, a link may fail and thus cease to transmit any data. Similarly, a router may fail and thus cease to transmit any data across a set of links, or all links incident to it. Finally, a router may crash and lose any volatile data that is required to operate efficiently within the network. In order to work around such failures, many network-layer protocols have nodes periodically check to make sure that their neighbors are active, and flood their newly-updated link states to the rest of the network. This updated information allows nodes in the network to revise their routing calculations when a link or router stops working properly. This updated link state information also helps a router that crashed to recover any volatile data it may have lost that is required for it to function efficiently in the network. Of course, any data sent along a non-functioning route must be retransmitted once a new route is calculated.

The higher-level transport layer is responsible for retransmitting data that did not arrive successfully at the destination because of a failure at the network layer. This facility is useful for recovering from failures that are slightly more subtle than a failed router or link. For example, a packet may be corrupted while being handled by a router or sent across a link (such corrupted packets must be detected by the transport layer, perhaps by validating the packet contents against a packet checksum). Additionally, two packets sent from a source to a destination may be delivered out-of-order. The transport layer is responsible for ordering packets correctly before delivering them to applications.

2.4 Byzantine Failures

Next, we consider Byzantine failures, a superset of the simple failures described above. A router with a Byzantine failure may exhibit arbitrary behavior. For example, it may calculate incorrect routes, fail to forward packets at random, or inject arbitrary packets into the network. A router may also fail maliciously and alter, fail to forward or impersonate packets from a particular source or to a particular destination discriminatorily. This arbitrary behavior can occur with data packets or with link state or neighbor packets. Finally, a router with a Byzantine failure may be even more subtle, pretending to be a dumb link between two other routers that usually acts as a passthrough but drops particular packets at random, or even colluding with other Byzantine routers to introduce widespread false information into the network.

It is immediately clear that a router with a Byzantine failure is capable of disrupting most or all of the network layer routing protocol components described above, causing the network not to

deliver packets with a high probability of success. The problemd definition for a routing algorithm with Byzantine robustness may therefore be formulated as follows: given a parameter f indicating the number of Byzantine nodes in the network, how many functioning paths are required between a source and a destination to guarantee that data can be delivered between them with high probability of success, in a reasonable time window?

2.5 Applications

Routing algorithms with Byzantine robustness are useful in any situation in which a certain network service level is required in an untrusted environment. In military settings, adversaries may be assumed to have control over routers, and such algorithms can be used to prevent denial of service attacks from such adversaries. In ad-hoc wireless and other peer-to-peer overlay networks, similar robustness guarantees may be useful. Finally, since the internet itself consists of many untrusted nodes, Byzantine robustness is a concern.

3 Algorithms with Byzantine Robustness

3.1 Robust Flooding

Each node is identified with both an address and a key pair for use in a public key cryptography scheme. Each node is manually preconfigured with its identity, its key pair, a parameter indicating the maximum number of nodes that can exist in the network and the maximum size of the data packets supported in the network. Finally, each node is also preconfigured with the identities and public keys of each of the t trusted nodes in the network as described below.

t nodes in the network are designated as trusted nodes. In addition to the manually preconfigured state mentioned above, each of the trusted nodes is also preconfigured with a list of node/public key pairs - one for each node in the network.

At runtime, each node stores some state in volatile storage. This includes a sequence number to use in the next data packet it sends, the latest public key list packet sent by each of the t trusted nodes, and the latest data packet sent by each of the nodes in the network.[1] Finally, each node stores an assortment of flags that are used to guarantee fair bandwidth access to each source in the network. Fairness is important if we want to enforce time bounds on communication (we do not elaborate any further on the fairness guarantees of these algorithms).

Signature verification is used to ensure that communication between nodes is successful. Since non-trusted nodes do not start out knowing the identities and public keys of other nodes in the network, the trusted nodes are responsible for distributing Public Key List (PKL) packets. PKL packets list the identity of the source; the sequence number of the source; the list of identities of all nodes known by that source, each with its public key; and the entire packet is signed with the trusted node's private key. PKL packets are sent by each trusted node periodically. When a node receives a PKL packet, it checks the validity of the digital signature with the preconfigured public key for the trusted source, and if the signature is valid and the sequence number is up to date, that public key list is installed in volatile storage (i.e. each node stores a copy of the latest PKL from each trusted node).

Data packets are similar to PKL packets. They contain the source identity, the source sequence number, the source public key, the destination address, the data, and a signature covering the whole packet. When a data packet is received by a node, its signature is verified using the source/public key listed in the packet (the source and public key must match a source and public key pair received in a valid PKL packet, and the signature must be valid for the data packet). If a data packet with a valid signature has a greater sequence number than the latest stored packet for that source/key pair, the in-memory copy of the latest data packet is updated and the packet is forwarded to every other neighbor.

[1] The latest data packet is not stored on just a per-node basis. The latest packet is actually stored for every (identity, public key) pair on the network. This is elaborated below.

Both PKL and data packets are ACKed, with retries for packets that haven't been ACKed. Furthermore, if a packet is received from source (s, k) with a sequence number less than the highest sequence number previously seen from (s, k), the later packet is sent back to the node that sent the outdated packet. This mechanism allows nodes that have restarted and lost their sequence number to recover it from the network so that they can continue to send packets. Packets with invalid signatures or packets that have an out-of-date sequence number are not forwarded.

Finally, there is a mechanism for a node to signal its neighbors that it has just restarted, and the neighbor will send a copy of the latest packet received from every other node to the newly-restarted node in response.

For a network in which there can be n nodes, this design requires $O(tn)$ storage at each node: the public key list entries for each of the t trusted nodes, plus the latest packet for each of the tn valid (identity, public key) pairs. It has the property that two nonfaulty nodes u and v can communicate if they are connected by at least one nonfaulty path, where every node on that path is connected via at least one nonfaulty path to a nonfaulty trusted node.

The cryptographic signatures on the packets prevent nodes from spoofing or tampering with packets. The reliable transport across each link (i.e. retries with ACKs that contain the signature of the original data packet) ensures that the data is actually transported across the link. Flooding ensures that each node transmits each packet to its neighbors, and this in turn ensures that packets will travel down nonfaulty paths if they do exist. Finally, faulty trusted nodes can transmit at most n (identity, public key) pairs, and nodes treat each pair as a valid source, ensuring that as long as each nonfaulty node in the network has received an up-to-date PKL from at least one nonfaulty trusted node, it can communicate with every other nonfaulty node that is reachable on a nonfaulty path.

3.2 Robust Link State Routing

This algorithm builds on the robust flooding algorithm to achieve a lower, but still reasonable level of robustness with better network bandwidth utilization.

Each node discovers its neighbors' identities. Each node then periodically floods a signed Link State Packet (LSP) to every other node in the network. The latest valid LSP from each distinct (source, key) pair is retained at every node to track link state. This information can then be used to generate valid routes. Note that storing link state per node increases the space requirement of the algorithm to $O(t^2n^2)$ at each node: we must now store up to tn link records for each of the tn identity/public key pairs in addition to the data stored in the robust flooding algorithm.

Routes are calculated at the source node, and included in data packets. Each data packet must be acknowledged by a cryptographically secure ACK from the destination node to the source. This way, the source can fail over to a new route if a previous route is found to fail (perhaps because of a Byzantine router on that path).

The Max Flow problem can be used to efficiently derive sets of node-disjoint paths between pairs of nodes. Assuming that there are f Byzantine routers in the network, any pair of nodes with at least $f + 1$ node-disjoint paths between them will be able to communicate by trying a sequence of such paths before finding one that works (the f Byzantine routers will cause at most f paths to fail).

The calculation of these paths is complicated if trusted nodes are Byzantine. In this case, the trusted nodes can introduce fake identity/key entries, and generate LSPs for such entries to make it look like these fake nodes exist and are present in the network. Even if we require that nonfaulty nodes check for link symmetry (if a reports a link to b, we only trust believe the link exists if b reports a link to a), a Byzantine trusted node may collude with a Byzantine regular node to make it look like another node exists. These fake nodes could make the calculation of node disjoint paths error prone or computationally expensive. To circumvent this problem, we can partition the graph into at most t instances: one instance for each distinct valid PKL stored at the source node. Within the set of identity/key entries listed in a single PKL (or set of consistent PKLs), it is tractable to calculate node-disjoint paths.

It is possible for a Byzantine node to act as a dumb relay, and forward neighbor identification messages and responses as though it is in fact a link. In order to prevent such a Byzantine node

from discriminatorily dropping packets transmitted between its two innocent neighbors, all packets are encrypted between neighbors using the receiver's public key. This way, the Byzantine node is forced to continue acting as a dumb relay: it cannot discriminate based on encrypted packet contents, and if it stops transmitting all packets or randomly sampled packets, the nonfaulty neighbors can simply mark the link as dead and therefore invalidate routes through the Byzantine node (note that it may be necessary for nodes to track link reliability over time in order to make such decisions).

Since packets are not flooded throughout the network, it is impossible for every node to know the latest sequence number seen from every other node/key pair. Thus it is impossible to update a newly-restarted node with the correct sequence number. Instead, each node must store a valid sequence number in nonvolatile memory so that it is not lost upon reboot. One mechanism that accomplishes this cheaply is to add a large constant x to the stored sequence number m every time a node reaches sequence number $m-1$. Thus if the node restarts, it will continue at sequence number $m+x$, which will be guaranteed to be larger than the previously-used sequence number $m+j$, $j < x$.

Of course, it is possible for a sequence number to wrap around, either legitimately (through extended use or as a result of frequent reboots) or because a Byzantine node jumps to a large sequence number early. In such cases, manual intervention is required to reregister the affected source with a new public key (via the trusted nodes) so that the node can continue to transmit data.

There are some interesting extensions to this link state algorithm proposed in [2]. The robustness guarantee of the robust link state algorithm is less comprehensive than the robustness guaranteed by the robust flooding algorithm. It would be possible to fail over to flooding for communication with a node that is unreachable on any of the node-disjoint paths calculated at the source: this would provide higher reliability in the presence of Byzantine nodes as necessary. It would also be possible to query intermediate nodes (using a higher-level network management protocol based on the transport layer) to identify where packets are getting dropped in a route. This would allow sources to identify candidate Byzantine nodes. Note that colluding Byzantine nodes could make it appear that an unfaulty node on the route between them is faulty in such a scheme. Finally, it would be possible to move to a more connection-oriented model, in which a source establishes a route to a destination up front, with a special packet that would register the route with every node on the path. Subsequent messages to the destination would be able to take that route without having to specify every hop explicitly in the packet header until the source deregisters the route. This reduces network bandwidth utilization per packet, but increases storage requirements to $O(t^3n^3)$ at each node: every source-destination pair would require storing $O(tn)$ state to track the registered routes.

4 Open Problems and Challenges

There are some limitations of the approaches described above that prevent their practical use in larger and more complex networks. For example, [2] assumes that links are exclusively point-to-point and not shared. When the transport medium is shared (e.g. ethernet and wireless networks), it is possible for a single node to interfere with communication from all other nodes. Depending on the specific technology in question, it may be difficult to isolate the faulty node to prevent it from interfering with the other nodes. The techniques mentioned above cannot be effective in such cases.

The storage requirements of the algorithms given in [2] are too large to be useful in networks consisting of many nodes. It is instead necessary to extend these protocols by adding levels of hierarchy to the network so that the state required per node is of tractable size. Both [2] and [1] propose such designs.

Perhaps the most challenging aspect of the algorithms given in [2] is the requirement for a public-key infrastructure to be in place allowing nodes to validate the primary key list packets disseminated by the trusted node service. In a real-world setting, such an infrastructure would have to support revocation checks, rollover, and key updates. These key management functions are difficult to implement securely in practice, and often prevent adoption of cryptography in real-world settings.

References

[1] Radia Perlman. Routing with byzantine robustness. Technical Report TR-2005-146, Sun Microsystems, September 2005. http://research.sun.com/techrep/2005/abstract-146.html.

[2] Radia Joy Perlman. *Network Layer Protocols With Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, October 1988. MIT/LCS/TR 429, http://hdl.handle.net/1721.1/14403.