Comparing performance of IPv6 multicast and unicast for software updates

Claudio Calvelli, Esther Payne and Brett Sheffield

May 9, 2022

Abstract

The librecast project states that "Multicast is, by definition, the most efficient way for multiple nodes to communicate". This experiment is designed to provide evidence of this efficiency by comparing multicast and unicast methods of sending the same data to a large number of nodes, as would for example happen when a software update is released.

We find that there is partial evidence to support this conclusion, and use the experience gained here to design future experiments which would provide a more definite answer.

1 Introduction

The first paragraph of RFC 3170 [1] states:

> IP Multicast will play a prominent role on the Internet in the coming years. It is a requirement, not an option, if the Internet is going to scale. Multicast allows application developers to add more functionality without significantly impacting the network.

There is a need for some experimental data to back these statements. We concentrate of measuring the impact of *software updates* on the network, because the proliferation of connected devices will make software updates a very important target for efficient use of the network, and because software updates are easy to simulate realistically by measuring the impact of copying a large file to a large number of nodes.

To provide evidence for the above statements, we compare the following methods of providing updates:

tcp: Traditional unicast using a TCP-based service: a server listens to TCP requests to send a copy of the software update, and each client requests the update from the server: this is the mechanism used by the vast majority of current services.

scp: Simple copy of the file using "scp" followed by a verification of the checksum; this is included mostly to confirm that the setup works correctly with standard, well-tested tools.

multicast: Full multicast: a number of servers provide the software update using multicast, and clients will obtain the updates by joining a multicast group and waiting for the data to arrive.

udp: Unicast using a UDP-based service: this is similar to the TCP case, but uses datagrams instead of virtual circuits: this mechanism is introduced because multicast is by necessity based on datagrams: there is no feedback from receiver to sender, and we want to help determine

which differences may be caused by unicast vs. multicast, and which ones by virtual circuits vs. datagrams.

Apart from the "scp" runs, do not use encryption in this experiment, all the three methods send the data unencrypted and verify that it has arrived correctly using a secure hash: this corresponds to the way some software updates are distributed, with an HTTP mirror providing the data and a secure hash provided by some more secure mechanism; we do not expect the results to be different when encryption is used for all transmissions, as used in many other cases, but we might consider a future experiment to test this.

Independently of the method selected, there are three "client" scheduling strategies:

immediate: All clients request updates at approximately the same time.

random: Each client will first wait a random time, up to the duration of the corresponding experiment using the "immediate" strategy.

random2: Each client will first wait a random time, up to twice the duration of the corresponding experiment using the "immediate" strategy.

We run the simulated software updates in a variety of network configurations and with a variety of file sizes to simulate the impact of different types of updates; in each case we measure network use, server load, client load and speed of update for each combination of update mechanism and scheduling strategy.

The rest of this report is structured as follows:

Section 2 describes the simplest possible network topology in which we can get useful measurements, and provides details on how we run the experiment.

Sections 3 and 4 describe two more network topologies, incresing the complexity and studying how different features affect the results.

Section 5 analyses the result of the experiment and compares the efficiency of unicast, multicast, and the transitional technology. There are also notes about testbed issues we identified, because anybody wishing to repeat the experiment will need to make sure they select an experiment testbed which is properly configured for IPv6 multicast.

Section 6 explores the possibility of further experiments, to make the simulation more realistic and more complete.

Appendix A provides some more details about the various programs which ran as part of the experiment, and where to find the full sources of all these programs.

Appendix B provides some important information to anybody who wants to repeat this experiment, and how to check if an experiment setup is configured as required.

2 "LAN" experiment

A number of clients (denoted by $\mathcal{C}$) request software updates from a single server; the server and all clients share a LAN so that updates have the shortest possible network path. This experiment will allow us to compare multicast and unicast in the simplest possible setting, and one which is possible on any existing local network in which IPv6 is enabled, and could represent for example distributing updates within an organisation.

A second experimental parameters indicates the size of the software updates as the number $\mathcal{N}$ of bytes contained in it. In the real world, clients may be more or less up-to-date so that each one may request a subset of all updates available; for this experiment we assume that all clients have all previous updates and are just requesting the latest one; a future exper-

iment may consider some more complex "real life" scenarios.

Figure 1 shows the network topology with $\mathcal{C} = 8$, i.e. there is a server sending data to 8 clients on a single LAN.

2.1 Experiment procedure

For a specified network topology (i.e. value for $\mathcal{C}$), and a list of update sizes (several distinct values for $\mathcal{N}$), we implement that network on an experiment testbed, then run a series of tests on it using all possible combinations of update size, update method and scheduling strategy. For organisational reasons, the testbed actually has 2 servers, the extra server does not take part in the update but directs the operations and collects results (we call this extra server the "director").

Each test starts with the director generating a file of the specified size filled with random data; this is copied to the server (a future experiment will use multiple servers, so generating the file on the director and copying it to all servers will make sure all send the same data, and in preparation for that we have this extra file copy instead of generating a random file directly on the server).

After generating and copying the file the director waits 60 seconds to make sure that the 1-minute load average of each node in the system is down to its baseline value; when we ran experiments without this wait, we had each run affecting the measurements of the next one, so it did not produce useful results.

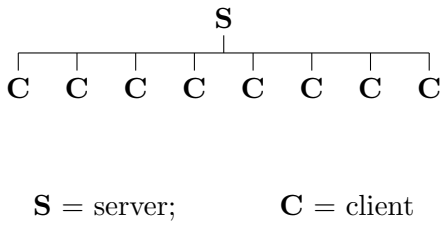


Figure 1: Network with8 clients on a single LAN

After the 60 seconds wait, the director asks the server node to start: this means starting two daemons, a resource monitoring tool and the update provider appropriate for the selected update method. These update providers are described below.

After the server has started, the director asks all client nodes to start as well; for the "random" and "random2" scheduling strategies each client will first wait a random time, this step is skipped for the "immediate" strategy; then each client starts a resource monitoring daemon identical to the one running on the server, and a client program to obtain the update.

When a client has successfully obtained the update, the monitoring daemon will record the time it has taken, finish another round of resource measurements, and sends all the data back to the director.

The director waits for all clients to have sent the data, then asks the server to stop, which will also trigger a copy of the server's resource measurements back to the director: all the measurements from all nodes are collected into a single "tar" archive and copied to one of our servers for later analysis.

The resource monitoring daemon records the following data every second:

- 1-minute load average as provided by the system
- user and system CPU time used by the whole system in the last second
- memory and swap use
- bytes sent and received on the network interface used to transfer the update data

Additionally, at the end of the experiment it also records the following data about the

update program itself (update provider for server, or the program obtaining the update for clients):

- Time elapsed between start and termination of the program, in milliseconds
- CPU time used by the program itself, in milliseconds
- CPU time used by the operating system to run the program, in milliseconds (this includes, for example, time used to obtain data from disk)
- The termination status: whether the program reported an error

The program running to provide the update method will also log some data via the same mechanism; unicast servers log the time of each request received and the time the corresponding reply has been sent completely: this allows a very precise count of the number of clients "active" (i.e. using server's resources) at any time, but it is not possible for multicast where the server only knows whether at least one client is active, or they are all inactive. The information about number of active clients is provided by the client logs in this case, and is sligtly less accurate.

Since the experiment procedure is automated by running a single program on the director, where possible we ran it many times on the same testbed, to have more experimental data without the extra overhead of setting up a new testbed.

We had several testbeds running this experiment, with a number of of clients between 20 and 51 depending on available resources at the time we started each run; these are the results identified by names like "S1L20" or "S1L51G" where the number following the "L" is the number of clients. Some of the testbeds only differed by a minor code change which resulted in no measurable difference, and the letters added as suffix helped us identify these. In all cases, results collected under the same name would have been running identical code on the same hardware.

As there were never a very large number of free nodes while we ran the experiment, we will need to leave a detailed study of scale to a future experiment.

2.2 Update methods

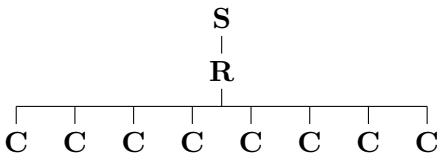
The "multicast" method uses the "IoT updater" demonstration program [2] to copy a file from server to clients: on the server side "iotupd" calculates the file's checksum, notifies the director that it's ready to send data, then runs a loop in which it sends the whole file and the checksum to a specified multicast group, then repeats the sending until asked to stop at the end of the experiment. Each client runs "iotupc" which waits for the data to arrive on the specified group and saves it to a local file, stopping when the file checksum matches. The only feedback from client to server is provided by MLD (Multicast Listener Discovery) messages citerfc:3810, which allow the server to know if there is at least one active client, or if they are all inactive: in particular, there is no mechanism for a client to request retransmission of missing data: the client just waits for the server to send it again. More details on the IoT updater can be found in appendix A.3 or in the source code [4].

The unicast "scp" method just uses the "scp" program (part of openssh) on each client to connect to the server and ask for a copy of the file; then it calculates the file digest and compares it with the digest provided by the director, repeating the whole thing in the unlikely case there is a mismatch: this allows us to show that the experiment setup is working with some well-known software; however the presence of encryption means that the re-

sults are not directly comparable with other experiments; a future experiment in which all methods use encryption will of course benefit from inclusion of this method. Instead of using the "ssh" daemon already running on each node, we chose to start another one on a different TCP port, so we can monitor precisely the server resource usage.

The unicast "tcp" experiment is similar in concept to the "scp" one, but implemented with code as similar as possible to the multicast case to make comparisons more meaningful. On the server side, the "iotup" program (in TCP server mode) calculates the file checksum, notifies the director that it is ready, then waits for connections; it replies to each connection with the file checksum and the full file data, sent as a single TCP stream. On the client side, the "iotup" program (in TCP client mode) connects to server, saves the file data, calculates the checksum and compares it with the one sent by the server; if the checksum match, the program exits with success, otherwise it retries the whole download. Because TCP already does its own verification, it is unlikely that the client will ever have to retry in this experiment when everything runs in the same building; however in a future larger and more distributed experiment we can expect to encounter network issues which will require retries from the client. The "iotup" program is described more fully in appendix A.4 and its sources are available in [5].

The unicast "udp" method is very similar to the "tcp" method, but uses UDP rather than TCP and that implies that retransmissions and duplicate detection need to be handled at the application level. On the server side, the "iotup" program (in UDP server mode) calculates the file checksum, notifies the director that it is ready for clients, then waits for requests coming in via UDP; these request specify a range of bytes to send, and the server replies with a sequence of UDP packets, each



S = server; **R** = router; **C** = client

Figure 2: Two LANs network with 8 clients

of which contain the same information as the packets sent using the "multicast" method. Each client starts by requesting the whole file (specifying both start and end offsets as 0, which the server interprets as "from 0 to end of file") and waiting for data: if there is no reply within a pre-determined time, it will retry the request. Once at least one packet arrives to the client, it will know the total data size and the file checksum, and will have part of the data. The client will wait until all the expected data has arrived, or a timeout occurs in which no new packets have arrived for a pre-defined time. The client will then decide whether any part of the file needs retransmitting and continue until it has received all the data it expects, and the file checksum matches. The "iotup" program is described more fully in appendix A.4 and its sources are available in [5].

3 "Two LANs" experiment

Similar to the previous experiment, but we investigate the effect of a multicast router in the network: there are two LANs connected together by a single router; the server is on the first LAN, and all the clients are on the second LAN. Like the previous experiments, the parameters are the number of clients $\mathcal{C}$, and the update size in bytes $\mathcal{N}$.

Figure 2 shows the network topology with

$\mathcal{C} = 8$, i.e. the same setting as the previous example (figure 1 on page 3) but with the clients separated from the server by a single router.

The experimental procedure is very similar to the previous experiment, we only describe the differences between them here.

After copying the update data to the server and waiting 60 seconds, the director will ask the router to start its own resource monitoring, and, for the multicast experiment, to start a multicast routing daemon. We used "lcroute" [6], which will form part of our own multicast routing platform. The director then waits for the router to signal that it has started and is ready to go.

After that, the experiment proceeds identically with the director starting the server and all clients and waiting for results. There is an extra step at the end: after the server has stopped, the director will ask the router to stop too, and waits for confirmation of this.

We only ran a 20 clients version of this experiment, when there happened to be free resources for this but not for anything larger. However we also had a time when one of the clients just failed to boot, so we ran it as a 19 clients experiment. Using a similar name scheme to the single LAN experiment, the results are identified as "S1R1L19C", "S1R1L19D" and "S1R1L20A" (the "R1" means that there was a single router in the network).

4 "Generic" experiment

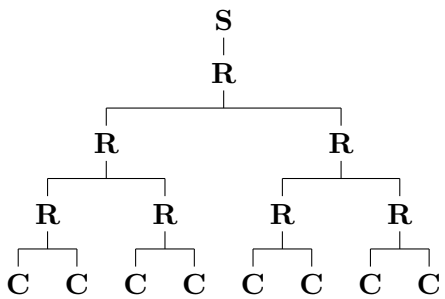
An extension of the previous ("Two LANs") experiment includes a longer network path between clients and server; for simplicity, and to generate the networks automatically, we specify a number of clients per LAN (denoted by $\mathcal{L}$) and the length of the network path indicated by the number of routers in the path, $\mathcal{R}$. The total number of clients will be $\mathcal{C} = \mathcal{L} * 2^{\mathcal{R}-1}$, and the routers form a tree structure.

A couple of examples will make this clearer: figure 3 shows the network topology with $\mathcal{R} = 3$ and $\mathcal{L} = 2$, so that the 8 clients are organised in 4 separate LANs, with 7 routers forming a tree structure with the server connected to the root of the tree. For comparison, figure 4 on page 7 shows the same number of clients arranged on 2 separate LANs ($\mathcal{L} = 4$) so that there are only 2 routers between each client and the server ($\mathcal{R} = 2$).

This is still a simplified view of a real system, but allows to extend the previous experiments to different circumstances. A future experiment might consider different networks.

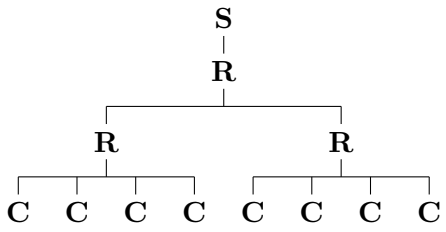
The experiment proceeds almost identically to the "Two LANs" experiment described above: there are more routers, but they are all set up in the same way as the single router on that experiment.

We ran experiments corresponding to both examples shown in the figures, using 20 clients rather than 8. We also ran a 40 clients version of the "3 router levels, 4 client LANs" experiment. Since there are 3 levels of routers, we called these experiments "S1R3L5B", "S1R3L5C" and "S1R3L10H"



S = server; **R** = router; **C** = client

Figure 3: Network with 8 clients, 2 clients per LAN



S = server; **R** = router; **C** = client

Figure 4: Network with 8 clients, 4 client per LAN

(the "L" indicates the number of clients per LAN, so in this case the total number of clients in the network is four times the number following the "L").

5 Experiment results

All the raw data produced by this experiment, as well as all the processed data referred to in this section is available [7]. In this section we show some results we can draw from the data. The appendices and software they cite describe how to convert the data to other formats, and how we processed it.

5.1 Run time

The first result we consider is the total time taken by each client to obtain the software update, showing separate averages for each network we ran and for each data size; also showing averages over all networks with the same number of clients, and over all experiments.

Table 1 on page 8 and the following tables show three representative set of results for single LAN experiments with 20, 40 and 50 clients respectively. Because of the way the number of clients has an effect on (unicast) servers, we do not average over all experiments, but only over experiments with the same number of clients. The complete set of tables for all the experiment sizes we have ran are available in the online data [7], and the appendix describes how to generate these and other tables from the raw data, also available online.

We use this data to consider the effect of scheduling on all update methods: multicast is largely unaffected, an update takes about the same time no matter when it's requested; on the other hand, all unicast method show quite a difference due to scheduling, with "immediate" being extremely slow, "random2" considerably faster, and "random" intermediate between the two. The conclusion we can draw is that having everything happening at once is bad for unicast, while multicast doesn't even notice the difference. We interpret this by remembering that unicast needs to send a separate data stream to each client, with overlapping requests competing for the available network bandwidth, while multicast sends a single stream; random scheduling reduces the overlap between clients compared to immediate, and so results in the client requiring less time to obtain the update, and random2 reduces the overlap even more with the corresponding result in terms of speed. We will return to this point later when analysing server resource usage.

Extrapolating from this (and hoping to confirm this with a future much larger future experiment), multicast would remain largely unaffected by adding more clients, while unicast would show an even more pronounced effect. In particular, the strategy of randomising updates to avoid overloading a server only works up to a limit: when it becomes impossible to schedule updates to avoid overlap, the only solution would then be to add servers, with the obvious environmental and economical costs; multicast on the other hand would continue to be considerate on resources, with any scheduling strategy.

UPDATE	SCHEDULE	SIZE	AVG	#DATA	MIN	MAX	STD
multicast	random	32	0.449	620	0.380	0.868	0.091
multicast	immediate	32	0.453	620	0.381	0.888	0.100
multicast	random2	32	0.459	620	0.381	0.879	0.111
tcp	random2	32	0.471	620	0.369	1.150	0.146
tcp	random	32	0.577	620	0.371	2.234	0.258
scp	random2	32	0.740	620	0.594	1.494	0.144
scp	random	32	0.855	620	0.593	1.946	0.232
udp	random2	32	1.476	620	1.380	2.215	0.135
udp	random	32	1.657	620	1.381	3.045	0.330
multicast	random2	128	1.824	620	1.517	9.841	0.712
multicast	random	128	1.881	620	1.517	12.862	1.047
multicast	immediate	128	1.893	620	1.515	12.301	1.179
tcp	random2	128	2.070	620	1.472	6.150	0.750
scp	random2	128	2.538	620	1.718	6.966	0.851
udp	random2	128	3.281	620	2.514	7.044	0.953
tcp	random	128	3.286	620	1.472	11.423	1.874
scp	random	128	3.684	620	1.721	12.421	1.690
tcp	immediate	32	4.159	620	0.819	5.724	1.096
scp	immediate	32	4.277	620	1.066	6.113	1.143
udp	random	128	4.629	620	2.515	13.075	2.071
udp	immediate	32	6.110	620	2.550	6.544	0.547
multicast	immediate	512	7.572	620	6.074	34.581	2.815
multicast	random	512	7.637	620	6.075	36.707	2.538
multicast	random2	512	7.784	620	6.061	34.731	2.785
tcp	random2	512	8.989	620	5.893	27.778	3.499
scp	random2	512	10.209	620	6.187	36.945	4.242
udp	random2	512	11.003	620	7.064	32.660	4.588
tcp	random	512	17.189	620	5.907	55.716	8.819
scp	random	512	17.998	620	6.194	54.247	9.278
scp	immediate	128	18.416	620	7.474	23.654	4.653
tcp	immediate	128	18.904	620	8.225	23.987	4.376
udp	random	512	20.256	620	7.070	45.046	9.178
udp	immediate	128	24.330	620	19.277	24.865	0.647
tcp	random2	2048	44.222	620	23.709	1:42.849	15.310
scp	random2	2048	45.096	620	24.122	1:58.827	17.116
udp	random2	2048	50.109	620	25.337	2:33.474	21.936
multicast	immediate	2048	51.380	620	24.510	3:44.979	35.377
multicast	random	2048	56.222	620	24.422	3:48.916	39.239
multicast	random2	2048	56.286	620	24.604	3:09.038	36.474
scp	immediate	512	1:15.397	620	29.420	1:34.834	18.204
scp	random	2048	1:16.874	620	24.173	5:04.713	40.983
tcp	random	2048	1:16.883	620	23.850	3:21.011	33.303
tcp	immediate	512	1:17.776	620	29.312	1:37.856	18.150
udp	random	2048	1:30.083	620	25.603	3:34.679	42.979
udp	immediate	512	1:37.484	620	1:33.639	1:38.341	0.629
scp	immediate	2048	5:07.019	620	1:59.498	6:19.772	68.494
tcp	immediate	2048	5:07.316	620	1:40.494	6:38.069	76.208
udp	immediate	2048	6:29.622	620	6:24.751	6:34.512	1.319

Table 1: Run time averaged on all 20-clients single LAN experiments

UPDATE	SCHEDULE	SIZE	AVG	#DATA	MIN	MAX	STD
tcp	random2	32	0.470	240	0.374	0.931	0.124
multicast	random2	32	0.501	240	0.390	0.788	0.066
multicast	immediate	32	0.516	240	0.391	0.788	0.084
multicast	random	32	0.517	240	0.391	0.782	0.087
tcp	random	32	0.599	240	0.375	1.407	0.268
scp	random2	32	0.892	240	0.730	1.394	0.131
scp	random	32	1.058	240	0.733	1.912	0.289
udp	random2	32	1.503	240	1.381	2.263	0.147
udp	random	32	1.638	240	1.382	2.693	0.292
multicast	random	128	1.991	240	1.576	3.170	0.296
multicast	random2	128	2.014	240	1.554	3.106	0.315
tcp	random2	128	2.457	240	1.477	7.682	1.256
scp	random2	128	2.744	240	2.125	4.797	0.493
multicast	immediate	128	2.970	240	1.602	9.050	2.249
udp	random2	128	3.411	240	2.518	6.849	0.864
scp	random	128	3.894	240	2.156	9.817	1.578
tcp	random	128	5.082	240	1.481	10.838	2.626
udp	random	128	5.245	240	2.519	12.182	2.155
multicast	immediate	512	9.728	240	6.166	23.921	3.419
tcp	random2	512	10.016	240	5.919	24.420	4.170
multicast	random	512	10.093	240	6.188	28.727	4.457
multicast	random2	512	10.237	240	6.229	23.909	4.137
tcp	immediate	32	10.380	240	0.441	11.378	0.905
scp	random2	512	11.174	240	7.769	24.677	3.206
udp	immediate	32	11.223	240	2.051	12.864	1.659
scp	immediate	32	11.948	240	3.155	14.004	1.852
udp	random2	512	12.913	240	7.091	39.998	6.948
scp	random	512	19.013	240	7.956	46.576	10.244
udp	random	512	24.379	240	7.229	57.849	13.038
tcp	random	512	24.829	240	5.934	52.507	12.517
tcp	immediate	128	44.933	240	38.600	46.676	1.090
udp	immediate	128	46.032	240	29.891	49.387	3.376
tcp	random2	2048	47.410	240	23.918	1:45.987	13.865
scp	random2	2048	48.819	240	30.105	1:55.842	15.277
udp	random2	2048	55.530	240	25.492	1:28.838	14.551
scp	immediate	128	57.878	240	40.066	1:01.547	2.729
multicast	random	2048	1:13.355	240	25.129	3:16.753	36.546
multicast	immediate	2048	1:13.490	240	25.303	3:11.828	37.083
multicast	random2	2048	1:15.799	240	24.757	3:14.033	40.015
scp	random	2048	1:32.879	240	30.903	4:26.519	54.277
tcp	random	2048	2:01.310	240	23.966	7:39.606	71.774
udp	random	2048	2:28.537	240	25.470	4:59.266	77.265
tcp	immediate	512	3:02.688	240	2:52.412	3:08.196	4.240
udp	immediate	512	3:07.592	240	2:27.987	3:14.868	8.560
scp	immediate	512	3:52.849	240	3:27.971	4:00.542	5.280
tcp	immediate	2048	12:06.111	240	11:45.811	12:32.999	14.514
udp	immediate	2048	12:29.976	240	9:40.448	12:57.216	29.157
scp	immediate	2048	15:38.172	240	14:42.846	16:02.758	18.548

Table 2: Run time averaged on all 40-clients single LAN experiments

UPDATE	SCHEDULE	SIZE	AVG	#DATA	MIN	MAX	STD
tcp	random2	32	0.495	100	0.372	1.017	0.152
multicast	random2	32	0.505	100	0.400	0.766	0.071
multicast	random	32	0.508	100	0.409	0.760	0.069
multicast	immediate	32	0.530	100	0.438	0.719	0.078
tcp	random	32	0.719	100	0.375	1.510	0.325
scp	random2	32	0.883	100	0.706	1.353	0.121
scp	random	32	1.034	100	0.735	1.804	0.246
udp	random2	32	1.522	100	1.382	2.395	0.175
udp	random	32	1.779	100	1.382	3.018	0.422
multicast	immediate	128	1.970	100	1.657	2.696	0.220
multicast	random2	128	1.999	100	1.618	2.777	0.234
multicast	random	128	2.002	100	1.682	2.814	0.215
tcp	random2	128	2.326	100	1.477	5.263	1.039
udp	random2	128	3.158	100	2.515	5.037	0.695
scp	random2	128	3.188	100	2.048	8.928	1.339
scp	random	128	4.363	100	2.047	8.534	1.610
udp	random	128	5.153	100	2.515	13.504	2.413
tcp	random	128	8.473	100	1.493	17.334	5.820
tcp	random2	512	9.075	100	5.915	19.407	3.381
multicast	immediate	512	9.336	100	6.421	17.812	2.638
multicast	random2	512	9.603	100	6.399	18.156	2.782
multicast	random	512	9.635	100	6.565	28.847	3.522
scp	random2	512	10.936	100	7.244	21.489	2.922
udp	random2	512	12.938	100	7.078	34.530	7.268
tcp	immediate	32	13.140	100	7.285	14.245	0.984
udp	immediate	32	14.391	100	1.763	15.720	2.423
scp	immediate	32	15.039	100	3.131	18.639	2.890
tcp	random	512	31.582	100	6.408	51.433	14.288
scp	random	512	39.688	100	8.419	1:55.657	21.096
tcp	random2	2048	48.119	100	23.848	1:20.777	12.697
scp	random2	2048	49.274	100	28.904	1:26.941	13.679
udp	random2	2048	53.634	100	25.450	1:23.572	14.387
tcp	immediate	128	55.297	100	48.837	57.866	1.241
scp	immediate	128	59.759	100	47.493	1:03.236	2.802
udp	immediate	128	1:03.299	100	6.841	1:21.072	16.380
multicast	immediate	2048	1:11.081	100	26.521	2:40.855	33.152
multicast	random	2048	1:13.406	100	25.710	3:34.842	37.505
multicast	random2	2048	1:13.918	100	26.110	3:14.669	37.021
scp	random	2048	1:19.381	100	28.874	2:09.298	22.424
udp	random	512	1:43.345	100	7.213	5:18.907	104.896
tcp	random	2048	2:01.977	100	26.221	3:28.729	47.149
scp	immediate	512	3:44.612	100	3:14.349	3:58.030	7.651
tcp	immediate	512	3:45.350	100	3:32.610	3:52.581	3.205
udp	immediate	512	4:10.318	100	3:30.503	4:25.560	12.961
udp	random	2048	12:32.300	100	33.267	23:18.408	495.582
scp	immediate	2048	14:46.974	100	13:46.491	15:40.753	22.995
tcp	immediate	2048	15:06.744	100	14:30.700	15:36.540	27.465
udp	immediate	2048	16:20.192	100	15:47.950	16:44.259	19.124

Table 3: Run time averaged on all 50-clients single LAN experiments

Moving to a different network topology, tables 4 on page 12 and 5 on page 13 show the same data collected from experiments with 3 levels of routers and with 20 and 40 nodes respectively (arranged as 5 and 10 clients per LAN, with 4 client LANs).

These tables show a different story, with multicast seeming to have became slower compared to the single LAN case, and in fact random scheduling is now visibly worse than immediate; while unicast seems to be unaffected by the presence of routers. However, the overall effect of scheduling on multicast is still less than on unicast.

There are two sources for the extra time taken by the multicast update. MLDv2 works on a LAN, so the routers need to forward this information before the servers know they need to start sending. And once the multicast data starts flowing, the kernel will need to query a routing daemon to know what to do with it. Since this extra delay is introduced when a client starts up, it is more visible with random scheduling: with immediate, whichever client happens to be fastest at starting up will see the delay while the rest don't need to. We will return to the issue of routing later when looking at router resource usage.

Extrapolating again from the data, we expect that increasing the number of clients will show that unicast continues to become slower when requests overlap, while multicast might show the opposite effect: as more and more clients send requests, there will be more overlap even in the case of random scheduling, and this would reduce the overall delay introduced by routing. A future experiment will need to look into this.

5.2 Server resources

Table 6 on page 14 shows the data sent by the server and the load level for each number of active clients during the multicast update experiment. The data is averaged over all experiments which used multicast update, file size 2GB and schedule immediate. The network data is measured in megabytes sent per second, and the load data is the fraction of CPU used.

By active client we mean a client which is in the middle of obtaining the file: for the multicast update, this is determined by comparing the "client starting to receive" and "client received the update" timestamps in each client's logs with the timestamps of each resource monitoring data item logged by the server.

We observe how the server load is mostly around 1 or just below, meaning that the server is using one processor almost constantly, with the rest being idle. The network data shows that it is sending at essentially the full bandwidth allowed by a gigabit network (i.e. 125 megabytes per second, or slightly less due to encapsulation overheads). Only the "0 clients" row show lower values. This corresponds to our expectation that the multicast server is unaffected by the number of clients receiving data, with the exception that it can stop sending if it knows that there are no listeners.

Table 7 on page 15 and the two following tables show similar data for the unicast update mechanism: scp, tcp and udp respectively. In these tables, the number of active clients is determined using server timestamps only, as the server logs the time it receives a request and the time it finishes sending the data.

Note that the udp table has been truncated to fit in the page: the full table is available with the online data [7]. The reason the table is longer than the others is due to the effect of packet loss and retransmission requests, where the client could request multiple retransmissions at once, and these are seen as independent requests by the server, so appear as multiple active clients. However, from the point of view of the server it is the number of requests being processed which determines re-

UPDATE	SCHEDULE	SIZE	AVG	#DATA	MIN	MAX	STD
tcp	random2	32	0.486	160	0.413	0.857	0.092
tcp	random	32	0.542	160	0.413	1.510	0.189
scp	random2	32	0.768	160	0.645	1.379	0.130
scp	random	32	0.845	160	0.645	2.404	0.265
udp	random2	32	1.640	160	1.382	2.685	0.394
tcp	random2	128	1.668	160	1.517	2.770	0.248
udp	random	32	1.740	160	1.382	3.400	0.506
scp	random2	128	2.215	160	1.767	4.531	0.534
scp	random	128	2.326	160	1.771	4.594	0.649
tcp	random	128	2.578	160	1.521	17.169	2.768
multicast	random	32	2.956	160	0.441	11.703	3.039
udp	random2	128	3.161	160	2.518	6.286	0.851
udp	random	128	3.633	160	2.518	16.711	2.049
multicast	random2	32	6.488	160	0.393	25.131	6.573
tcp	random2	512	7.050	160	5.950	25.702	2.866
scp	random2	512	7.855	160	6.253	17.925	1.997
tcp	random	512	8.353	160	5.957	31.286	4.580
scp	random	512	8.650	160	6.249	29.032	3.499
udp	random2	512	9.221	160	7.144	20.484	2.426
multicast	immediate	32	9.988	160	0.437	26.879	5.060
udp	random	512	13.309	160	7.140	1:21.003	12.776
udp	immediate	32	13.684	160	9.929	16.720	1.745
scp	immediate	32	19.089	160	5.695	25.429	4.030
tcp	immediate	32	19.778	160	9.382	24.546	3.710
multicast	random	128	22.378	160	2.637	2:08.551	29.449
multicast	random2	128	24.699	160	2.669	1:47.464	28.941
multicast	immediate	128	26.194	160	2.714	2:37.730	31.942
tcp	random2	2048	31.810	160	23.755	56.947	9.548
multicast	random	512	32.992	160	10.865	1:11.537	13.079
scp	random2	2048	33.270	160	24.203	2:10.262	14.753
tcp	random	2048	35.501	160	23.770	1:11.214	12.898
scp	random	2048	38.515	160	24.216	2:24.038	19.189
udp	random2	2048	40.809	160	25.716	1:30.622	15.079
udp	random	2048	43.804	160	25.754	1:28.552	15.946
udp	immediate	128	53.284	160	40.613	1:03.909	4.391
multicast	immediate	512	1:00.312	160	21.049	2:52.877	49.635
multicast	random2	512	1:02.547	160	10.931	6:21.248	80.396
scp	immediate	128	1:20.516	160	33.407	1:43.557	17.224
tcp	immediate	128	1:26.070	160	39.478	1:44.015	14.855
multicast	random2	2048	2:55.555	160	43.934	14:35.734	115.476
multicast	random	2048	3:14.968	160	52.113	8:56.891	113.232
udp	immediate	512	3:39.251	160	3:09.716	4:29.492	22.626
multicast	immediate	2048	4:26.618	160	1:00.310	28:13.926	371.051
scp	immediate	512	5:25.786	160	2:38.441	6:36.335	58.170
tcp	immediate	512	5:48.382	160	2:31.774	7:02.183	52.129
udp	immediate	2048	14:32.426	160	13:22.372	17:57.721	96.870
tcp	immediate	2048	22:20.067	160	10:26.054	28:36.747	267.532
scp	immediate	2048	23:43.940	160	10:25.981	27:54.636	248.291

Table 4: Run time averaged on all 20-clients multi-LAN experiments

UPDATE	SCHEDULE	SIZE	AVG	#DATA	MIN	MAX	STD
tcp	random2	32	0.529	160	0.417	1.053	0.131
tcp	random	32	0.625	160	0.415	1.458	0.226
scp	random2	32	0.910	160	0.774	1.328	0.115
scp	random	32	1.134	160	0.776	3.334	0.465
udp	random2	32	1.648	160	1.384	2.703	0.336
udp	random	32	1.721	160	1.383	2.810	0.320
tcp	random2	128	2.052	160	1.528	4.750	0.734
tcp	random	128	2.248	160	1.522	5.009	0.798
scp	random2	128	2.797	160	2.204	6.048	0.653
scp	random	128	3.399	160	2.224	11.074	1.444
udp	random2	128	3.607	160	2.519	11.073	1.361
udp	random	128	5.564	160	2.521	12.905	2.408
multicast	random	32	6.101	160	0.566	24.720	6.823
tcp	random2	512	7.762	160	5.965	15.036	2.174
multicast	random2	32	8.111	160	0.488	30.969	8.136
scp	random2	512	9.790	160	7.879	15.236	1.417
tcp	random	512	10.979	160	6.028	34.501	6.644
udp	random2	512	11.126	160	7.160	24.528	4.161
udp	immediate	32	11.528	160	3.074	18.874	3.038
multicast	immediate	32	12.011	160	1.360	32.142	7.767
scp	random	512	14.551	160	7.841	1:02.594	9.782
tcp	immediate	32	15.037	160	4.398	28.530	9.016
scp	immediate	32	15.088	160	5.020	26.625	6.889
multicast	random2	128	21.569	160	2.025	1:43.286	27.547
udp	random	512	27.277	160	7.210	1:23.328	15.368
multicast	random	128	33.158	160	6.106	2:39.887	43.681
multicast	immediate	128	38.772	160	6.485	2:45.259	42.853
scp	random2	2048	44.471	160	30.617	1:11.273	8.053
tcp	random2	2048	44.721	160	23.951	1:17.464	10.409
udp	immediate	128	46.402	160	20.692	1:07.467	8.137
tcp	random	2048	46.485	160	24.500	1:38.500	11.869
scp	random	2048	47.482	160	30.638	1:25.759	11.886
udp	random2	2048	1:00.357	160	25.598	2:34.283	19.983
tcp	immediate	128	1:01.996	160	26.066	1:54.735	34.578
scp	immediate	128	1:06.825	160	37.021	1:50.726	26.815
multicast	immediate	512	1:08.845	160	35.259	2:59.909	51.933
multicast	random	512	1:22.918	160	7.913	6:24.232	101.206
udp	random	2048	2:01.485	160	26.239	5:14.096	68.661
multicast	random2	512	3:06.481	160	8.334	9:13.771	174.107
udp	immediate	512	3:07.410	160	2:28.131	3:48.856	15.271
tcp	immediate	512	4:08.877	160	1:53.864	7:53.519	134.009
multicast	immediate	2048	4:11.318	160	38.832	13:16.754	176.408
scp	immediate	512	4:41.463	160	2:39.828	7:33.659	124.255
multicast	random	2048	7:05.852	160	2:17.409	27:13.674	394.802
multicast	random2	2048	8:29.905	160	2:17.898	31:41.704	450.464
udp	immediate	2048	12:51.869	160	10:33.090	15:53.358	57.179
tcp	immediate	2048	18:07.755	160	7:30.619	33:59.081	668.873
scp	immediate	2048	20:11.670	160	10:42.087	36:31.129	598.695

Table 5: Run time averaged on all 40-clients multi-LAN experiments

	Network TX					Server load				
#CLIENTS	AVG	#DATA	MIN	MAX	STD	AVG	#DATA	MIN	MAX	STD
0	20.9	1683	0.0	180.0	43.3	0.8	2135	0.1	1.6	0.3
1	113.4	2782	0.0	223.9	25.0	0.9	3156	0.2	1.4	0.1
2	116.3	1674	0.0	226.4	22.2	1.0	2102	0.2	1.4	0.1
3	118.9	970	22.9	222.4	9.5	1.0	1477	0.2	1.4	0.1
4	120.1	1040	0.0	220.5	9.8	1.0	1586	0.3	1.4	0.1
5	118.9	489	24.1	218.2	12.0	1.0	1125	0.3	1.4	0.1
6	117.3	546	0.0	122.9	12.1	1.0	1198	0.2	1.4	0.1
7	115.5	460	0.0	225.1	11.0	1.0	1171	0.3	1.4	0.1
8	121.8	1319	41.1	122.9	3.9	1.0	2190	0.3	1.4	0.1
9	113.9	342	34.4	226.0	11.3	0.9	1123	0.3	1.4	0.1
10	112.9	1745	0.0	122.9	28.6	0.9	2588	0.2	1.4	0.1
11	119.2	719	0.0	211.3	9.0	0.9	1509	0.3	1.4	0.1
12	119.0	504	76.8	122.9	5.5	0.9	1337	0.3	1.4	0.1
13	117.0	294	96.6	223.0	10.3	0.9	1191	0.2	1.4	0.1
14	116.6	277	63.5	218.8	11.1	0.9	1287	0.2	1.4	0.1
15	115.5	199	32.0	217.2	10.7	0.9	1126	0.2	1.4	0.1
16	112.1	340	63.4	222.6	8.7	0.9	1237	0.3	1.4	0.1
17	115.5	147	66.0	217.2	11.1	0.9	1011	0.2	1.4	0.1
18	114.3	206	97.7	211.3	8.7	0.9	1196	0.3	1.4	0.1
19	112.1	8670	94.1	122.9	1.9	1.0	9689	0.2	1.4	0.1
20	122.4	2006	93.9	122.9	2.1	0.9	3234	0.2	1.4	0.2
21	116.0	135	74.4	122.9	6.5	0.9	957	0.3	1.0	0.1
22	115.2	98	0.0	225.4	17.7	0.8	885	0.3	1.0	0.1
23	115.0	99	65.2	122.9	7.7	0.8	988	0.3	1.0	0.1
24	113.7	116	94.5	122.9	7.2	0.8	1063	0.3	1.0	0.1
25	115.4	102	93.6	122.9	6.5	0.8	1036	0.3	1.0	0.1
26	115.8	80	76.9	217.2	13.7	0.8	954	0.4	1.0	0.1
27	115.0	75	93.8	122.9	6.1	0.8	1186	0.3	1.0	0.1
28	116.8	78	105.0	122.9	5.0	0.8	938	0.3	1.0	0.1
29	116.5	83	107.3	122.9	4.5	0.8	956	0.3	1.0	0.1
30	116.3	187	93.4	122.9	4.2	0.8	993	0.3	1.0	0.1
31	112.9	65	76.1	122.8	6.0	0.8	710	0.3	1.0	0.1
32	106.4	69	93.7	122.9	10.0	0.8	815	0.3	1.0	0.1
33	117.1	26	94.2	219.7	21.9	0.8	542	0.3	1.0	0.1
34	114.6	39	105.7	122.9	4.1	0.8	635	0.3	1.0	0.1
35	113.4	20	93.8	122.9	7.2	0.8	567	0.3	1.0	0.1
36	114.2	24	104.6	122.9	5.9	0.8	647	0.3	1.0	0.1
37	112.9	45	22.1	122.9	14.9	0.8	923	0.3	1.0	0.1
38	116.3	198	44.5	222.3	10.1	0.8	1248	0.3	1.0	0.1
39	115.6	298	101.5	227.0	18.3	0.7	1727	0.3	1.0	0.1
40	112.3	810	0.0	122.9	17.0	0.7	2257	0.3	1.0	0.1
41	119.1	65	23.8	122.9	13.0	0.7	1035	0.3	1.0	0.1
42	122.5	280	104.7	122.9	2.2	0.7	1356	0.3	1.0	0.1
43	119.3	25	107.2	122.9	5.8	0.7	670	0.3	1.0	0.1
44	117.8	15	105.4	122.9	6.8	0.7	610	0.3	1.0	0.1
45	103.5	9	23.5	122.8	30.6	0.7	435	0.3	1.0	0.1
46	102.9	10	18.9	122.9	34.0	0.6	424	0.3	1.0	0.1
47	108.8	7	83.5	122.8	17.2	0.6	482	0.3	1.0	0.1
48	114.2	162	107.5	122.9	2.7	0.6	685	0.3	1.0	0.1
49	113.0	289	102.9	223.8	18.9	0.6	764	0.3	0.9	0.1
50	111.9	55	108.2	122.8	3.2	0.6	278	0.3	0.8	0.1
51	122.8	331	122.7	122.9	0.0	0.5	625	0.3	0.8	0.1

Table 6: Server resource data: multicast

	Network TX					Server load				
#CLIENTS	AVG	#DATA	MIN	MAX	STD	AVG	#DATA	MIN	MAX	STD
0	0.1	1647	0.0	57.8	1.8	1.3	1751	0.2	3.3	0.8
1	65.0	273	0.0	123.2	30.1	2.1	273	0.2	3.5	0.9
2	120.7	299	0.0	240.9	17.8	2.4	299	0.2	3.9	0.8
3	101.8	128	0.0	240.6	39.4	1.8	128	0.3	4.0	1.0
4	115.0	166	0.0	123.5	23.6	1.7	166	0.1	3.8	1.1
5	119.0	486	0.0	123.5	14.2	1.2	486	0.1	3.8	0.5
6	119.8	492	0.0	123.5	12.1	1.3	492	0.2	4.0	0.4
7	118.3	174	0.0	123.5	19.1	1.4	174	0.2	3.5	0.6
8	93.2	634	59.4	123.5	25.1	1.0	634	0.2	3.8	0.5
9	107.6	433	22.4	123.5	24.0	1.1	433	0.1	4.0	0.6
10	101.4	886	30.3	123.5	25.7	1.1	886	0.1	3.5	0.5
11	97.1	905	0.0	123.5	30.9	1.0	905	0.1	3.5	0.6
12	79.1	1072	30.9	123.5	39.3	0.9	1072	0.1	3.3	0.6
13	89.2	996	2.8	237.4	35.0	0.9	996	0.1	4.0	0.5
14	61.5	1654	21.4	123.5	33.1	0.7	1654	0.1	3.7	0.5
15	68.1	1564	24.1	123.5	44.4	0.7	1564	0.1	4.0	0.6
16	68.1	1322	21.3	123.5	45.7	0.7	1322	0.1	3.7	0.7
17	47.5	3420	22.4	123.5	37.6	0.7	3420	0.1	4.0	0.5
18	49.1	5399	20.8	123.5	35.4	0.6	5399	0.0	4.0	0.5
19	72.7	9051	20.3	123.5	47.7	1.2	9051	0.0	3.9	1.1
20	49.0	20734	19.4	123.5	40.1	0.5	20734	0.0	3.9	0.4
21	120.6	208	28.5	123.5	11.8	1.7	208	0.2	3.7	0.4
22	117.4	40	55.3	123.5	14.0	1.9	40	0.3	4.0	0.7
23	122.7	148	88.7	123.4	3.2	1.8	148	0.3	3.9	0.5
24	117.5	20	81.3	123.5	11.3	2.1	20	0.3	3.9	1.1
25	121.3	76	90.1	123.5	6.5	2.1	76	0.3	4.0	0.7
26	121.9	108	76.0	123.5	5.9	1.7	108	0.3	3.7	0.6
27	122.3	150	68.3	123.5	5.5	1.4	150	0.8	3.9	0.5
28	116.4	26	82.6	123.4	13.3	2.2	26	0.3	4.0	1.1
29	121.1	89	62.8	123.5	8.9	1.9	89	0.3	4.0	0.9
30	123.0	186	83.6	239.9	9.8	1.6	186	0.3	4.0	0.7
31	116.4	24	91.2	123.5	11.8	2.4	24	0.3	4.0	1.0
32	116.5	32	64.5	123.5	13.3	2.6	32	0.2	4.0	1.0
33	120.7	123	93.9	123.5	7.4	2.3	123	0.3	3.8	0.7
34	122.2	190	72.7	236.8	10.8	1.7	190	0.2	4.0	0.7
35	121.6	174	64.3	240.0	12.1	2.1	174	0.3	4.0	0.6
36	121.9	401	78.4	123.5	5.7	1.8	401	0.5	4.0	0.7
37	118.1	75	54.3	240.0	19.9	2.1	75	0.2	4.3	0.9
38	120.0	540	75.8	123.5	8.8	1.8	540	0.2	4.1	0.4
39	116.3	208	52.0	123.5	13.2	2.0	208	0.2	4.3	0.8
40	95.2	8300	26.4	123.5	7.2	2.5	8300	1.0	9.1	1.1
41	122.7	131	118.9	155.5	3.0	2.3	131	1.0	4.3	0.6
42	125.8	60	101.8	240.1	21.6	2.8	60	0.3	4.7	1.1
43	122.3	56	82.1	240.2	17.3	3.0	56	0.2	4.7	1.2
44	123.9	68	118.7	240.4	14.4	2.5	68	0.3	4.7	1.1
45	122.8	823	60.8	123.5	2.2	1.8	823	0.3	4.5	0.5
46	122.5	456	70.3	240.3	6.7	1.8	456	0.3	4.5	0.9
47	122.6	617	77.3	123.5	3.2	2.0	617	0.2	4.4	0.6
48	122.9	5497	18.0	240.3	5.8	3.4	5497	0.5	12.0	1.7
49	122.0	11019	14.6	240.7	19.8	3.8	11019	0.3	14.1	1.8
50	122.9	2258	35.9	123.5	4.0	3.2	2258	0.3	11.9	1.8
51	122.5	5350	120.6	123.3	0.6	1.5	5350	0.3	2.5	0.4

Table 7: Server resource data: scp

	Network TX					Server load				
#CLIENTS	AVG	#DATA	MIN	MAX	STD	AVG	#DATA	MIN	MAX	STD
0	0.7	2552	0.0	121.8	7.6	0.2	2968	0.0	1.1	0.2
1	85.8	223	4.8	240.6	42.2	0.2	421	0.0	1.1	0.2
2	118.3	262	8.9	123.5	19.1	0.3	465	0.0	1.1	0.2
3	113.9	92	13.1	123.5	27.0	0.2	298	0.0	1.1	0.2
4	120.8	296	17.9	240.8	15.5	0.3	501	0.0	1.1	0.2
5	120.1	442	22.2	123.5	10.9	0.2	645	0.0	1.0	0.1
6	120.8	527	19.6	240.6	13.9	0.2	734	0.0	1.0	0.2
7	111.0	205	31.1	123.4	17.4	0.2	407	0.0	1.0	0.1
8	88.2	541	0.0	123.4	29.3	0.1	743	0.0	1.0	0.1
9	93.3	597	22.8	123.3	24.7	0.1	798	0.0	1.0	0.1
10	97.9	1081	21.7	123.4	32.1	0.1	1281	0.0	1.0	0.1
11	98.9	953	34.5	123.4	25.2	0.1	1154	0.0	1.0	0.1
12	89.2	1007	8.1	123.3	38.0	0.1	1209	0.0	1.0	0.1
13	84.7	1176	23.5	123.7	41.6	0.1	1379	0.0	1.0	0.1
14	64.3	2068	26.7	123.4	36.1	0.1	2273	0.0	1.0	0.1
15	66.0	1468	23.9	123.4	39.7	0.1	1665	0.0	1.0	0.1
16	71.8	2013	20.6	123.4	44.2	0.1	2214	0.0	1.0	0.1
17	66.2	2268	20.8	123.5	43.3	0.1	2470	0.0	1.0	0.1
18	41.7	7083	19.6	123.5	32.4	0.1	7286	0.0	1.0	0.1
19	78.7	7429	20.3	240.6	47.4	0.1	7622	0.0	1.0	0.1
20	47.2	20724	18.6	123.4	38.3	0.1	20857	0.0	1.0	0.1
21	122.1	262	4.6	123.2	9.4	0.1	344	0.0	1.0	0.1
22	123.0	158	119.8	123.3	0.5	0.1	239	0.0	1.0	0.2
23	121.4	13	107.2	123.2	4.4	0.3	93	0.0	1.0	0.2
24	122.5	36	112.4	123.5	2.1	0.3	118	0.0	0.7	0.1
25	123.0	73	122.2	123.2	0.2	0.2	155	0.0	0.7	0.1
26	123.1	118	121.9	123.4	0.1	0.2	201	0.0	0.7	0.1
27	123.0	125	111.9	123.2	1.0	0.2	203	0.0	0.7	0.1
28	121.8	20	112.2	123.1	3.3	0.3	101	0.0	0.6	0.1
29	122.5	191	32.1	123.2	6.6	0.2	275	0.0	0.7	0.1
30	114.9	10	79.2	123.5	17.2	0.3	94	0.1	0.7	0.1
31	123.0	46	121.6	123.2	0.3	0.2	125	0.1	0.6	0.1
32	122.7	241	53.9	123.2	4.5	0.2	321	0.1	0.7	0.1
33	122.9	313	120.4	123.2	0.4	0.2	397	0.0	0.7	0.1
34	122.2	95	118.5	123.5	0.6	0.2	177	0.0	0.7	0.1
35	122.8	31	119.9	123.2	0.7	0.2	113	0.0	0.7	0.1
36	122.7	56	116.1	123.2	1.1	0.2	139	0.0	0.7	0.1
37	122.8	54	119.9	123.4	0.6	0.2	137	0.0	0.6	0.1
38	122.4	248	117.4	123.2	1.0	0.2	330	0.0	0.7	0.1
39	121.9	641	115.3	123.5	0.8	0.2	725	0.0	0.7	0.1
40	121.9	6164	114.2	123.5	1.5	0.3	6236	0.0	0.7	0.1
41	122.6	372	114.9	240.8	8.7	0.2	438	0.0	0.7	0.1
42	118.4	215	111.4	123.2	4.0	0.2	277	0.0	0.7	0.1
43	120.2	330	109.3	123.5	3.3	0.1	395	0.0	0.7	0.1
44	120.5	416	104.7	123.5	1.9	0.2	479	0.0	0.7	0.1
45	120.5	163	105.3	240.6	10.2	0.2	225	0.1	0.7	0.1
46	116.8	234	109.0	123.5	4.8	0.2	295	0.1	0.7	0.1
47	120.0	485	104.8	123.5	4.0	0.3	550	0.1	0.7	0.1
48	122.5	6064	99.5	240.6	3.7	0.3	6124	0.0	0.7	0.1
49	123.1	9945	102.1	240.9	18.7	0.2	9988	0.0	0.7	0.1
50	121.0	2540	100.7	123.5	5.2	0.2	2570	0.0	0.6	0.1
51	118.6	5785	99.1	123.2	4.9	0.2	5799	0.0	0.6	0.1

Table 8: Server resource data: tcp

	Network TX					Server load				
#CLIENTS	AVG	#DATA	MIN	MAX	STD	AVG	#DATA	MIN	MAX	STD
0	2.8	2493	0.0	122.3	14.6	8.4	43434	0.0	39.1	5.0
1	49.9	2264	0.0	123.2	37.3	8.6	7400	0.1	39.1	5.1
2	51.0	1755	0.2	240.1	36.4	8.6	8918	0.1	39.1	4.9
3	47.0	1207	0.3	123.2	37.1	8.6	9976	0.1	39.1	4.8
4	40.9	765	0.4	123.2	37.2	8.6	10865	0.1	39.1	4.8
5	41.9	462	0.5	123.2	39.2	8.7	11783	0.1	39.1	4.7
6	47.6	333	1.4	123.2	40.7	8.8	13204	0.1	39.1	4.8
7	57.5	300	1.4	123.2	41.1	8.7	14851	0.1	39.1	4.8
8	64.6	205	0.8	123.2	41.0	8.8	17162	0.1	39.1	4.7
9	69.1	203	4.0	240.3	39.9	8.8	20404	0.1	39.1	4.8
10	65.7	192	0.5	123.2	39.3	8.9	24192	0.1	39.1	4.8
11	66.2	259	5.1	123.2	35.0	9.0	28190	0.1	39.1	4.7
12	64.0	257	5.2	123.2	34.8	9.0	48106	0.1	39.1	4.7
13	97.5	108	9.3	123.2	28.9	9.2	10068	0.1	44.3	4.9
14	94.7	73	6.4	240.3	36.1	9.2	9896	0.1	44.3	4.9
15	86.9	79	5.4	123.2	35.9	9.3	9758	0.1	44.3	4.9
16	97.1	95	8.8	123.2	32.8	9.3	9784	0.1	44.3	4.9
17	93.1	128	7.1	123.3	34.6	9.3	10216	0.1	44.3	4.9
18	91.0	147	5.9	123.2	36.1	9.4	10631	0.1	44.3	4.9
19	106.9	281	0.5	123.2	31.2	9.3	11490	0.1	44.3	4.9
20	122.5	19842	2.5	123.2	4.3	8.5	31627	0.1	44.3	3.8
21	86.9	108	6.2	123.2	35.8	9.4	12824	0.1	44.3	4.9
22	83.9	137	4.9	123.2	36.0	9.4	13980	0.1	44.3	4.9
23	92.6	165	6.9	123.2	32.9	9.4	14908	0.1	44.3	4.9
24	80.3	137	5.4	123.2	40.2	9.2	17750	0.1	48.1	4.9
25	96.5	110	14.0	123.2	31.1	9.8	8510	0.1	48.1	4.9
26	97.3	90	1.6	123.2	34.4	9.7	8551	0.1	48.1	4.9
27	103.6	99	2.8	123.2	29.5	9.8	8449	0.1	48.1	4.9
28	101.1	84	25.1	240.6	37.1	9.8	8572	0.1	48.1	4.9
29	96.9	97	2.6	239.8	36.1	9.8	8757	0.1	48.1	4.9
30	90.2	112	3.5	123.2	38.2	9.9	9035	0.1	48.1	5.0
31	96.4	124	3.7	123.2	38.1	9.8	9388	0.1	48.1	5.0
32	98.0	134	5.0	123.2	34.6	9.9	9674	0.1	48.1	5.0
33	100.8	152	5.6	123.2	32.5	9.8	9877	0.1	52.2	5.0
34	100.2	135	3.5	240.3	37.0	9.8	10205	0.1	52.2	5.0
35	104.9	146	4.4	123.2	30.2	9.7	10455	0.1	52.2	5.0
36	107.5	176	2.2	123.2	27.0	9.5	11670	0.1	52.2	5.0
37	112.4	168	22.7	240.4	26.3	10.2	7569	0.1	52.2	5.0
38	117.0	219	35.3	123.2	15.6	10.2	7583	0.1	52.2	5.0
39	117.4	351	18.0	123.2	17.1	10.2	7822	0.1	52.2	5.0
40	122.1	6534	3.8	239.9	5.1	9.5	14127	0.1	52.2	4.7
41	97.1	92	4.0	123.2	29.8	10.3	7718	0.1	52.2	5.0
42	103.2	81	26.4	123.2	25.5	10.3	7806	0.1	52.2	5.0
43	104.0	95	36.1	240.1	29.4	10.4	7749	0.1	52.2	5.1
44	106.1	114	1.3	240.3	31.6	10.4	8001	0.1	52.2	5.2
45	101.9	90	12.1	123.2	27.1	10.3	8018	0.1	52.2	5.0
46	102.6	116	19.3	240.3	34.0	10.2	8166	0.1	52.2	5.1
47	107.7	104	28.4	240.3	28.5	10.3	8216	0.1	53.4	5.1
48	114.9	179	20.6	240.3	27.0	10.3	8701	0.1	53.4	5.2
49	123.0	9757	33.9	240.4	19.1	17.7	16402	0.1	53.4	8.1
50	121.7	1127	36.9	123.3	8.8	11.9	7869	0.1	53.4	6.5
51	122.7	13551	28.8	123.2	2.0	14.8	20244	0.1	53.4	5.8
			...					...		

Table 9: Server resource data: udp

source usage, rather than the number of distinct clients producing these requests, so we decided to leave the data as it is. Some of the more recent experiments have had the client code modified to always serialise the retransmission requests, but we did not re-run all experiments as there appeared to be no reason to do so.

The unicast tables surprised us a bit. We expected to see the network constantly saturated as all streams are competing for it, except of course when there are no active clients, in which case the server would not send anything. Instead, it appears that the server isn't always saturating the network when sending to multiple clients, and that explains how some testbeds seemed to take even longer than expected to complete the file copy. We are not sure how to explain this, unless there is some hardware bottleneck for example obtaining the data from disk: for multicast, which reads a file sequentially from start to end in a single thread, the kernel's readahead may be helping, while the unicast servers have multiple threads all reading the same file at different positions, and it may be able to defeat the kernel's caching. However we cannot say for sure until we repeat these experiments measuring other resources such as disk I/O bandwidth.

For server load, scp shows a drop in processor utilisation which appears to match the drop in network utilisation; however where the network is fully saturated the load is normally above 1, and growing with the number of clients: this would reflect the number of processors busy encrypting data, and we note that the corresponding tcp data shows very low load, reflecting the fact that all the server is doing is moving bytes from disk to network card.

Udp server load appears to be very high although it follows a similar pattern to the scp data. Since there is no encryption and the server is just moving bytes from disk to network card, this seems to be a higher processor utilisation than one would expect. It is possible that the network stack is optimised for TCP and doesn't handle UDP very well; we have also observed high levels of packet loss at times which aren't justified by the network setup. Possibly running these experiments on different hardware or using a different operating system may show a different result. All we can do with these numbers is show them and offer our guesses.

5.3 Routing

We are not specifically analysing multicast routing in this experiment, however a glance at some router network data will explain why the various multicast updates appear to take longer to start, and this will affect the random update schedules more than immediate. We consider just two examples with 3 levels of routers and a path going from the server to the "leftmost" client LAN (in the tree representation used elsewhere in this document). In the specific example this goes through routers numbered 6, 4 and 0.

Table 10 on page 19 shows the network traffic in and out of a router while a multicast immediate experiment is running. Table 11 on page 20 is the same but for a longer experiment (larger file). The column headers refer to the router number, the router's interface (towards clients or servers) and the data direction as seen from the router: therefore "6:client:tx" would show data sent by router 6 towards the clients and "0:server:rx" shows data received by router 0 from the direction of the server (it receives it from router 4, not directly from the server, but it is data arriving "from the direction of the server"; the interface names we have used here do not imply direct network connection to server or clients).

This shows the delay introduced by MLDv2 [3] and multicast routing. The server starts transmitting data (as shown in the first

TIME	6:server:rx	6:client:tx	4:server:rx	4:client:tx	0:server:rx	0:client:tx
15	0.000	0.000	0.000	0.000	0.000	0.000
16	0.000	0.000	0.000	0.000	0.000	0.000
17	31.821	31.481	0.000	0.000	0.000	0.000
18	102.929	102.925	96.945	0.000	0.000	0.000
19	102.943	102.938	102.661	0.000	0.000	0.000
20	91.908	92.244	96.470	0.000	0.000	0.000
21	91.200	91.192	90.498	0.000	0.000	0.000
22	91.540	91.554	91.267	0.000	0.000	0.000
23	91.677	91.668	91.407	0.000	0.000	0.000
24	91.718	91.723	91.389	0.000	0.000	0.000
25	91.786	91.767	91.275	0.000	0.000	0.000
26	92.054	92.076	91.807	0.000	0.000	0.000
27	92.087	92.084	91.696	0.000	0.000	0.000
28	92.125	92.108	91.886	79.955	54.679	54.503
29	92.482	92.496	92.227	92.249	92.436	92.122
30	92.506	92.509	91.929	91.965	92.375	92.109
31	92.567	92.572	92.371	92.334	92.386	92.070
32	91.928	91.919	92.065	92.062	92.609	92.324
			...			
121	97.112	96.773	93.653	93.621	93.968	93.680
122	103.307	103.308	102.290	102.329	100.069	99.610
123	103.110	103.111	102.908	102.877	103.113	102.948
124	103.296	103.296	102.949	102.970	103.169	102.679
125	103.425	103.417	103.073	103.047	103.094	102.815
126	103.550	103.554	103.209	103.227	103.383	103.026
127	103.205	103.207	102.923	102.920	103.426	103.103
128	103.215	103.220	102.844	102.893	103.218	103.027
129	103.321	103.313	103.047	102.992	103.252	7.776
130	103.474	103.476	103.029	103.039	103.355	0.000
131	103.344	103.346	103.107	103.094	103.353	0.000
132	103.275	103.274	103.014	103.012	103.399	0.000
133	103.249	103.244	103.002	103.066	103.278	0.000
134	103.220	103.223	102.864	102.824	103.193	0.000
135	103.194	103.189	102.934	102.896	103.241	0.000
136	103.557	103.560	103.059	103.080	103.340	0.000
137	103.292	103.292	103.159	103.168	103.477	0.000
138	103.287	103.292	102.954	102.942	103.294	0.000
139	103.281	103.281	102.983	102.995	103.273	0.000
140	103.599	103.601	103.044	103.052	103.354	0.000
141	103.517	103.515	103.257	103.251	103.710	0.000
142	103.401	103.403	103.136	103.134	103.353	0.000
143	103.563	103.561	103.301	103.288	103.634	0.000
144	103.742	103.369	103.027	103.073	103.285	0.000
145	103.430	103.414	103.102	103.089	103.428	0.000
146	103.351	103.354	103.047	103.029	103.420	0.000
147	103.446	103.446	103.036	103.052	103.337	0.000
148	53.905	54.235	91.459	91.378	103.464	0.000
149	0.000	0.000	0.000	0.000	16.790	0.000
150	0.000	0.000	0.000	0.000	0.000	0.000
151	0.000	0.000	0.000	0.000	0.000	0.000
152	0.000	0.000	0.000	0.000	0.000	0.000

Table 10: Example routing

TIME	6:server:rx	6:client:tx	4:server:rx	4:client:tx	0:server:rx	0:client:tx
0	0.000	0.000	0.000	0.000	0.001	0.000
1	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000	0.000	0.000
3	0.000	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000	0.000	0.000
5	0.036	0.031	0.016	0.011	0.002	0.000
6	0.000	0.000	0.000	0.000	0.004	0.000
7	0.000	0.000	0.000	0.000	0.000	0.000
8	0.000	0.000	0.000	0.000	0.000	0.000
9	0.000	0.000	0.000	0.000	0.000	0.000
10	0.000	0.000	0.000	0.000	0.000	0.000
11	0.000	0.000	0.000	0.000	0.000	0.000
12	0.000	0.000	0.000	0.000	0.000	0.000
13	66.604	66.594	26.243	25.867	0.000	0.000
14	122.845	122.863	103.362	103.362	66.075	66.074
15	122.741	122.743	102.924	102.969	103.100	103.132
16	122.757	122.753	103.973	103.973	102.937	102.958
17	122.753	122.745	102.971	102.971	103.281	103.286
18	122.773	122.758	102.589	102.591	102.615	102.581
19	122.730	122.745	102.634	102.633	102.223	102.231
20	122.719	122.739	102.580	102.584	102.201	102.185
21	122.765	122.753	102.915	102.912	102.587	102.589
22	122.740	122.747	102.735	102.736	102.439	102.456
23	122.774	122.745	102.968	102.967	102.585	102.600
24	122.727	122.747	102.791	102.789	102.517	102.503
			...			
536	122.757	122.728	103.281	103.281	102.987	0.000
537	122.749	122.745	103.131	103.130	102.741	0.000
538	122.738	122.749	103.585	103.586	103.073	0.000
539	122.715	122.741	103.480	103.483	103.184	0.000
540	122.739	122.734	103.409	103.409	103.267	0.000
541	122.731	122.741	103.340	103.341	102.992	0.000
542	122.879	122.854	103.256	103.252	103.014	0.000
543	122.707	122.749	103.418	103.419	102.915	0.000
544	122.775	122.745	121.826	6.456	69.119	0.000
545	122.753	122.718	123.076	0.000	0.000	0.000
546	122.735	122.762	123.201	0.000	0.000	0.000
547	122.731	122.732	123.081	0.000	0.000	0.000
548	122.782	122.756	123.193	0.000	0.000	0.000
549	122.743	122.749	123.080	0.000	0.000	0.000
550	122.737	122.762	123.190	0.000	0.000	0.000
551	122.723	122.765	123.076	0.000	0.000	0.000
552	122.790	122.706	123.194	0.000	0.000	0.000
553	122.722	122.745	123.080	0.000	0.000	0.000
554	122.762	122.751	123.198	0.000	0.000	0.000

Table 11: Example routing

data column, network data received by the "top" router) but this data only starts being forwarded with a small delay; and longer delay are introduced by subsequent routers until it eventually gets to the client(s).

Later in the experiment, the server is still sending, because some clients is still listening. However forwarding only goes part of the network path we are following in this table, meaning that all clients in the specific client LAN at the end of this path have received all the data, they are no longer listening, and the routers are no longer forwarding there; however data is still forwarded for part of the path because other LANs served by a prefix of this path are still requiring data.

6 Future work

Due to time limitations we have only measured network performance for a small set of regular network topologies, corresponding to the example networks shown in sections 2 to 4, using more clients than shown in the figures; of course the real world is made up of rather more irregular topologies and it it would be interesting to investigate more variations in this area in a future set of experiments.

We also limited the experiments to about 50 clients, as we have been unable to allocate larger networks: the testbed never had enough free resources. To properly test how the various method scale, we would need some experiment setup where we can easily allocate 500 or more nodes and have them running for days.

Alternatively, we would like to run the experiment distributed across several sites to have a much larger number of total nodes, and also a more representative network structure. However the unicast experiments are likely to require massive amount of network bandwidth, and the multicast experiments require proper multicast configuration at all sites and multicast routing between then, so these factors will limit the choice of sites.

We also simplified the software update by assuming that all clients request exactly the same file, rather than a more complex situation in which every client requests a different subset of all available updates, due to its own unique update history; unicast of course will need very little change: since each client receives its own separate stream, they don't have to contain the same data; for multicast, we already have (in the present experiment) servers only sending data when somebody is listening, so it would never need to send all possible updates all the time, and we expect multicast to handle this situation well; however without a corresponding experiment, "expect" is all we can say about this.

All the experiments we ran included a single server. The main point of this experiment was to show that a single server is sufficient to provide updates for a large number of clients using multicast, while unicast will require multiple servers in this case. However, there are reasons other than server and network load why one would want multiple servers, for example reliability: if the single, extremely efficient, server has a fault, the updates stop; ideally, these multiple servers will be reachable by completely different network paths as well. We think that multicast will help with that too, for example multiple servers can each send data at a fraction of the bandwidth, and when they all work clients will get the advantage of the combined output from all servers, while a network or server failure would automatically result in a corresponding reduction of speed, and the subsequent recovery or replacement of the faulty parts would automatically result in the system returning to full speed. This claim, of course, needs a separate experiment to justify.

Another type of network activity which can benefit from multicast is live streaming, where the server will only need to send the stream

once; this case is similar to software updates and probably does not need a separate experiment; however if several choices of bandwidth and quality are required the situation is different. In the unicast case it's obvious how the sender can provide different quality streams to different clients, for multicast the simplest answer is to provide several streams with different quality, with the client subscribing to the one which best match its requirements: this would save network and server resources but there may be better way of achieving this result, for example using layered codecs to send only one copy of the lowest quality stream, then a second stream with the difference between that and the next highest quality. We don't know at present if these codecs would involve more server resources than the re-encoding required to provide multiple stream with different quality, but in any case we would find it useful to run another experiment to measure these costs and compare them with the expected savings in terms of network usage.

For this experiment we transmitted all data unencrypted between the nodes and used a secure hash to determine whether it was received correctly. This corresponds to a traditional situation in which HTTP mirrors provide the data, and a checksum is provided over a more secure mechanism for verification. More recently, most systems are moving to HTTPS with the added overhead of encryption on every communication: we expect that the benefits of multicast shown in this experiment will continue to apply, but we have not tested this, and will consider a future experiment in which we extend the multicast update method to add encryption of all communication, comparing this with the normal stream encryption used for example with HTTPS.

A Programs

To run each experiment we had to implement a network topology on an experiment testbed, set up each node in the testbed, run the experiment itself and collect the results; additionally, we had to analyse the results of groups of experiments together. This appendix describes the programs used for all various tasks, and includes references to where the full source code can be found for the programs we developed.

A.1 Preparing a testbed and running an experiment

The programs described here and other useful tools are in the experiment setup repository [8] under the `bin` *directory for the programs to run on the local system and the* `objects` *directory for the programs to run on the testbed.*

Given the number of servers, clients and routers (if appropriate to the experiment) we developed a simple program to generate action files for "jfed" [9] so that the process could be automated; a single program "mknet" provided action files for all the experiment topologies described in this report by providing appropriate options; for the networks shown as examples in the figures we just ran:

```
local$ mknet L=8
local$ mknet R=1 L=8
local$ mknet R=3 L=2
local$ mknet R=2 L=4
```

As can be seen, omitting "R" results in a single LAN network in which the number of clients is specified by "L" for consistency with the other networks (where it indicates the number of clients on each client LAN).

By default, the program generates an experiment name indicating the parameters provided: for the four examples above this would be: "S1L8", "S1R1L8", "S1R3L2" and "S1R2L4"; the data generated will be stored

in a directory inside `/var/tmp` named after the experiment (the name of the experiment starts with "S1" to indicate the number of servers, in preparation for a future multi-server experiment, and the directory can be changed with other command-line options).

One of the files generated, `action.yaml`, is suitable for using as argument to the `-a` (action file) option to the "jfed-cli" tool and will provision the testbed; the program also generates `action.rspec` which is suitable for using with the "jfed-gui" tool. We do not describe these tools here as they are provided by fed4fire, but see [9].

Once the testbed is up and running, we need to copy some things to it, for example the actual programs which will run on it and information about the experiment to run. The list of data sizes is also specified at this point, for example to run with 32, 64 and 512 megabytes on the first single LAN experiment defined above ("S1L8")

```
local$ setup-experiment 32,64,512 S1L8
```

This sets up the "director" node and copies the `objects` directory of the repository to it. To complete the setup, one needs to connect to it and then run a program there:

```
local$ ssh-experiment S1L8 director0
director0$ /tmp/experiment/setup-all
```

The testbed is now ready to run the experiment by running a program on director0 and specifying a "rsync" destination where it will copy the data it collects:

```
local$ ssh-experiment S1L8 director0
director0$ cd /tmp/experiment
director0$ ./run-experiment \
  --rsync DESTINATION
```

The destination must be set up as appropriate to receive the data. Other options to "`run-experiment`" are normally not required; such options are documented in the script itself.

If required, the "`run-experiment`" program can run many times to obtain more data without additional setup overhead; no need to repeat any of the previous steps.

Internally, the "`run-experiment`" program calls other programs running on the director, but also on servers, routers and clients as necessary to implement the procedure described in sections 2 to 4. These programs have names like "`start-tcp-server`" or "`start-multicast-router`" to start what is required for a particular experiment on a particular node (in this example, start the TCP experiment on a server, and start the multicast routing daemon on a router, respectively). All these programs are found in the repository cited.

One issue we found while developing programs to automate the experiment is that network interface names may be different when booting different testbeds; each node has two interfaces, a control interface used by the testbed administration, as well as to log in to it from outside the experiment, and a second interface connected as required by the experiment's network topology and used to transfer the data files during the experiment; for a router node, there are obviously more interfaces: therefore we needed to determine what interface name was actually assigned to what. The fed4fire documentation mentions a tool to obtain the necessary information on each node, however the tool did not work: it required a version of Python which is no longer available, and all it produced for us was a syntax error. Instead of debugging that, it was easier for us to have the "setup-all" script figure out what interface is going to be used for what based on the MAC address listed in the output from the jfed program, and configure them as required: this may be different for each node. It also sets up the monitoring so that the interfaces

are reported in a consistent way, as is important when looking at network usage data for routers.

A.2 Resource monitoring

The program described here is in the lwmon repository [10].

There are many monitoring tools for Unix system, however in our experience they tend to use more resources than programs which do the actual work, or else they are designed to sample information only once a minute, which is not enough for this experiment.

Because of experience using other monitoring tools and not finding one which we actually want to use on a live system, we have developed our own over the years, which we call "lwmon", for Light-Weight system MONitoring, which, as the name suggests, is very considerate in its use of resources and can safely run very frequently without impact on the system. It can also report on its own resource usage, so one can confirm that it is, indeed, light-weight.

Without going into complete details, each node sets up its own configuration for lwmon, which then measures memory, swap, cpu and network usage and system load every second, its own resource usage every 10 seconds. The appropriate program (update provider for servers, routing daemon for routers if requires by the experiment, and the program which gets the updates for clients) also runs as a child process of lwmon, so the latter can report on the resource used by this program.

A lwmon configuration for a client is shown in figure 5 on page 25. This means that load average, memory/swap usage and network usage are sampled every second, lwmon's own resource usage every 2 seconds, will run the appropriate program for a particular experiment (in this case, update via multicast), and save the information into a file in a compact binary format. The second column of the lines specifying what to measure is the name used to report it.

The program running does not depend on the scheduling strategy selected, as that is handled before: we do not want to measure the time it takes to sleep for a random duration, only the time it takes to download the update. The file name for the results, on the other hand, contain the scheduling strategy (in this case, random) and the update size (64 megabytes), as we need to keep things distinct.

Another thing to note is that the "network" line asks to monitor the interface using its systemd name (enp6s0 in this example) because that's how it will be able to find it in the system, but reports it using the name "if1" as this is the interface name we have used in the action file and rspec.

The configuration for a server is essentially identical, with "client" replaced by "server", and for a router it is very similar, as shown in figure 6 on page 25. The network interface reported as "if1" has network traffic going to or from clients, and "if2" has traffic going to or from servers.

Once the "`start-...`" program terminates, lwmon automatically reports on its resource usage and the wall clock time it has taken to run, then runs one more round of measurements and exits. For clients, the program normally terminates when it has obtained the update successfully; for servers and routers the program terminates when the director signals the end of the experiment.

The program which generates the lwmon configuration file and calls lwmon will wait for it to terminate, then copies the file it produced back to the director node. This allows the director to collect all the data about the experiment in one place.

As mentioned, lwmon produces a file containing data in a packed binary format. To just look at the data, the tool has an option to read that binary file back in and produce a

```
hostname client2

load lavg 1
cpu cpu 1
memory memswap 1
network if1 1 enp6s0
self self 2

program exp 5 /tmp/experiment/start-multicast-client enp6s0
print /tmp/results/client2.multicast.random.64 overwrite binary
```

Figure 5: lwmon client configuration

```
hostname router0

load lavg 1
cpu cpu 1
memory memswap 1
network if1 1 enp4s0
network if2 1 enp6s0
self self 2

program exp 5 /tmp/experiment/start-multicast-router enp4s0
print /tmp/results/router0.multicast.random.64 overwrite binary
```

Figure 6: lwmon router configuration

human-readable output:

```
$ lwmon -P - \
  -R data/router0.multicast.random.64
```

There is a separate tool which reads one or more data files and produces SQL statements which can be used to import it into a database, for example:

```
$ lwmon-to-sql [options] FILES | \
  sqlite3 database.sqlite
```

As described below in A.5, for this experiment we have developed a tool which reads the lwmon data directly to create summaries, and uses a sqlite database as cache to speed up operation; this is not the full data as produced by `lwmon-to-sql` but instead just the data needed to generate tables and graphs like the ones shown in this report.

A.3 Multicast update method

The software for this experiment is from the IoT Updater example described in [2], and whose sources are in [4].

The multicast server uses MLD snooping to decide whether there is at least one active client, and starts sending as soon as one is present, stopping when it knows no client is still requiring data. While running, it simply sends the file to an IPv6 multicast group continuously on a loop. The file is sent as a sequence of datagrams each containing some data from the file, as well as a checksum for the whole file, the size of the whole file, and the size and starting file offset for the data contained in the datagram.

The client node joins the multicast group and immediately begins receiving the file. When the first datagram is received, the client creates a memory-mapped file of the appropriate size and begins writing each datagram at the appropriate offset in the file. When enough data has been received, the client starts a checksumming thread to verify the data on disk matches the expected checksum. If it does, the client parts the multicast groups and exits. If the checksum does not match, the client continues receiving and writing data until it does. Data payloads received are compared to the data on disk before copying and only different data is written or counted towards received data.

The IoT Updater can interleave data sent across multiple multicast groups, and use the sequence numbers in packets to detect packet loss. If packet loss is due to congestion, the client can join fewer groups to lower the received data rate. This can function as a basic form of flow control. This function was not used during the experiments - the client and server were set to use only a single multicast group

A simple way to run the server manually to send a file to multicast group `ff1e::42` via interface `eth0` would be:

```
iotupd sourcefile ff1e::42 eth0 --mld
```

To receive the data one runs the client as:

```
iotupc destfile ff1e::42 eth0
```

Note that the client does not specify how to find the server: this is not Single Source Multicast and there is no need to provide this information.

If there are routers between the server and the client, these routers need to be able to forward multicast traffic. We used lcroute [6] on a Linux node, starting lcroute as:

```
lcroute eth0 eth1
```

Where `eth0` is the interface where the multicast traffic will be sent to (i.e. where the unicast routing towards the client goes) and `eth1` is the interface where the multicast traffic will

arrive from (i.e. where the unicast routing towards the server goes). The version of lcroute we used needed to know this, however this restriction will be lifted in future.

Note that the kernel needs to be configured for IPv6 multicast routing for this to work. A simple way to check is to run:

```
sysctl net.ipv6.conf.all.mc_forwarding
```

This may reply 0 or 1 on a kernel with a suitable configuration (it will only show 1 if lcroute or another multicast routing daemon is running). If the kernel is not configured appropriately, the command will produce an error.

A.4 Unicast update methods

The software used for this experiment (which includes UDP and TCP) can be found in [5].

For both update methods, the server waits for client requests, then forks and handles each request in a separate process. Clients send a request using the appropriate protocol, then wait for a reply.

The programs for both server and client, and for both UDP and TCP, are started with the same command, with different parameters:

```
iotup server|client udp|tcp \
FILENAME SERVER_ADDRESS
```

The server will open the file specified and binds a socket to the address specified; the client will send a request to the server address specified, and saves the file to the name indicated.

For TCP, clients open a stream connection to a predefined port on the server; the server accepts connection and sends back a header containing the file size and a checksum of the whole file, then the file data from beginning to end. The client simply reads all the data from the server, saves it to a file, and checks that the size and checksum are correct: if they aren't, it will repeat the whole request.

For UDP, clients send a datagram to a predefined port on the server, containing a range of file offsets to send: this initial request will be for the "whole file", specified by a start and end offset both zero (for the end offset, zero will be interpreted by the server as "until end of file"). The server replies by sending a sequence of data packets to the source address of the datagram, each packet containing the total file size, file checksum, some file data as well as the data size its offset within the file. This means that the client can have all the required information even in case of packet loss.

The client waits for data from the server, rememebring which data it has received and which one has not arrived. If all the data arrives, it calculates the checksum and decides whether it has received the file correctly, retrying if it has not.

A timeout makes sure that the client does not wait forever in case of packet loss: if no data arrives for a predefined period, and some data is still missing, the client assumes that the server has stopped sending and requests retransmission of the packets it knows it has not received, then goes back to waiting for data.

Note that this behaviour from the client can result in the server seeing more active clients than real clients in case of packet loss: clients may time out while the server is still sending, request retransmission which will be seen by the server as a new client: this is the reason why the UDP results collected by number of clients can show more active clients than real clients.

If routers are present, they do not need anything particular as long as unicast routing between server and clients is set up correctly; we still ran our monitoring on routers to collect network bandwidth and other information.

A.5 Summarising data

The programs described here are in the "experiment scripts" repository [8].

The "lwmon" program (appendix A.2) produces a lot of data, as it measures several items every second, and of course each node in the experiment produces its own set of measurements. The end result is that we have tens of millions of data items divided into thousands of "tar" archives, each one produced by a single experiment run; inside these archives, files are named after the node which produced them (for example, client1 or router0) and the actual experiment ran (for example, multicast or tcp) but other information is only in the name of the archive itself, so the first step is to extract all these files renaming them so that the names contain all the necessary information (this situation is because when we started running the experiments, only the "director" node had all the information, but the file names were generated by the nodes creating the files: in a future experiment we are planning to change this).

These tar archives have names like:

```
S1L20C-vwall1-20220119101508-multicast-
random-2048-20220131154314.tar.gz
```

this example has been produced by the network we named "S1L20C", booted on virtual wall 1 when the time on our local system was 19/Jan/2022 at 10:15:08; this archive contains the results for the "multicast" method with "random" scheduling and a data size of 2048MB; the run ended on 31/Jan/2022 at 15:43:14 (local time of the testbed). The actual timestamp values are not important per se, but together with the name we assign to the network they generate unique names for each run (the name has been folded into multiple lines to fit in the layout of the report, but of course it is a single string).

The `averages` script reads the lwmon data directly and produces summaries as described below; however this can be slow so it uses a database to cache intermediate results. This database could also be useful for other processing. There is a table indexing tarballs and two other tables for different types of cached summaries.

The table indexing tarballs, "tarballs" has the following columns:

```
id              int
filename        varchar(256)
filesize        int
mtime           int

topology        varchar(256)
suffix          varchar(32)
testbed         varchar(32)
boot_time       int
update_method   varchar(256)
schedule        varchar(256)
file_size       int
run_time        int
n_servers       int
router_levels   int
n_clients       int
```

The first group of columns are the file information: "id" is the internal unique identifier used in the cache database, "filename" is the tarball's filename, and "filesize" and "mtime" are provided by the filesystem and are used by the program to update the cached data if the file appears to have changed (this can happen if we run the program while results are still arriving from the testbed).

The second group of columns are obtained by splitting the file name into its components, and interpreting the testbed name to determine the number of clients and routers.

The decoded lwmon data for a particular element is accumulated in one of two tables, depending on whether the element depends on the number of active clients or not. The "data" table contains data which does not depend on the number of clients and contains:

```
tar_id          int
```

```
dataname        varchar(256)
count           int
min             float
max             float
total           float
square          float
```

The "count", "total" and "square" contain a summary of the data collected, which could appear multiple times in the same tarball Each data item can be present multiple times in the same tarball (for example, each client could contribute one element); these are all added together in the "total" column, and their squares are added in the "square" column; when selecting a tarball for processing, we can use this information to calculate averages and standard deviations. The "min" and "max" columns record the minimum and maximum data item found, respectively.

The other table, "summary", contain information grouped by number of active clients; it has one extra column, "n_clients".

The "averages" script produces averages from a specified subset of experiment runs, and also maintains the cache database. The general usage is:

```
$ averages [options] DATA_NAME \
  CACHE_DATABASE \
  TARBALL [TARBALL]... | less
```

This will probably produce a lot of data, so it's best to send the output to a file or pipe into a pager.

Currently, "DATA_NAME" can be one of "run-time", "server-tx" and "server-load" described below, and a sample output is shown in figures 7 and 8 on page 30. The program determines the number of active clients by combining information about client start and stop times and adds that to the table.

We have currently defined three data types, but the script allows adding more very easily:

run-time: Total time the client required from sending the first request to server until it received the complete file.

server-load: Server load by number of active clients; this is sampled every second or more often if there is a change in the number of active clients, and is the total number of CPU millisecond reported used divided by the milliseconds since the previous sample.

server-tx: Data sent over the network per second; this number is sampled every second or more often if there is a change in the number of active clients, and is the number of bytes sent divided by the number of milliseconds since the previous sample, and further divided to obtain a number of megabytes per second.

A pre-filled cache database is available online together with the raw data [7]

For routing data, we wanted to see the effect of multicast routing protocols by comparing the data going into a router with the data leaving the router; it makes less sense to average over multiple runs as each experiment proceeded at a different pace, and averaging only served to hide the changes in the data streams. Accordingly we used a different script, "`router-data`" to collect bandwidth utilisation data from several routers (in a single run) and produce a multi-column output showing the utilisation against the time since the start of the run. The usage is:

```
router-data SPEC [SPEC]... TARBALL
```

where each "SPEC" is an interface specification of the form "router:interface:direction" in which "router" is the number of the router (counted from 0 for the leftmost client LAN in the tree diagrams shown in this document,

```
$ ./bin/averages -p run-time ~/DB/summary.sqlite ~/results/S1L49F-
UPDATE    SCHEDULE  SIZE          AVG #DATA          MIN          MAX      STD
multicast random      32        0.502   539        0.403        0.767    0.069
multicast random2     32        0.507   539        0.406        0.793    0.077
multicast immediate   32        0.512   539        0.406        0.795    0.083
tcp       random2     32        0.516   539        0.372        1.182    0.186
tcp       random      32        0.733   539        0.372        1.970    0.381
scp       random2     32        0.885   539        0.735        1.858    0.138
scp       random      32        0.974   539        0.742        2.129    0.207
...
scp       immediate 2048    15:01.874   539    14:01.121    15:31.685   12.585
udp       immediate 2048    15:30.195   539    14:32.472    15:48.049   15.527
```

Figure 7: Example experiment result

```
$ ./bin/averages -p server-tx ~/DB/summary.sqlite ~/results/S1L49F-
UPDATE    SCHEDULE  SIZE #CLIENT        AVG #DATA        MIN        MAX    STD
scp       immediate  128      24        0.0     1        0.0        0.0      -
tcp       immediate  128      10        0.0     1        0.0        0.0      -
tcp       immediate  512      21        0.0     1        0.0        0.0      -
tcp       immediate  512      31        0.0     1        0.0        0.0      -
multicast immediate   32       2        0.0     2        0.0        0.0    0.0
multicast immediate  128       7        0.0     2        0.0        0.0    0.0
...
udp       immediate  512      23      180.3     2      120.1      240.6   85.2
udp       immediate   32      41      180.3     2      120.1      240.6   85.2
tcp       immediate 2048      41      180.6     4      120.3      240.8   69.5
```

Figure 8: Example experiment result

and proceeding left to right in the diagram until reaching the rightmost router, then continuing one level up, etc. Therefore the 4 client LANs in an experiment with 3 levels of routers are connected to routers 0 to 3, then router 4 joins routers 0 and 1, and so on.

An example output for the command:

```
router-data 6:server:rx 6:client:tx \
4:server:rx 4:client:tx \
0:server:rx 0:client:tx \
S1R3L10H-vwall1-20220309204654-\
multicast-immediate-2048-\
20220311052100.tar.gz
```

is shown in figure 10 on page 19 (truncated to fit in the page) and is available in the online data [7] (look for run number S1R3L10H-20220311052100).

The "make-online-summaries" script in [8] contains many examples of analysing data, as well as showing how the summaries included in the online data [7] have been generated.

B Important notes on repeating this experiment

IPv6 requires multicast, however when preparing to run this experiment we found that some testbeds did not actually support it.

It is very easy to check that multicast packets are forwarded between nodes: build "mcastsend" and "mcastread" from mcast-tools [11], select two nodes in an experiment which are connected to the same LAN, and run code like the one shown in figure 9 on page 32, where the interface names in the command must be changed to reflect the actual network interface used on these nodes.

The expected result is that both examples show a lot of data stored in `/tmp/datafile`, as multicast packets are forwarded from the sender to all nodes which request receiving them. However on some testbeds we found that this was not the case, and instead produced the result shown in the figure: some multicast groups were forwarded, while others were blocked, and this means that any part of the experiment which uses groups which are blocked will not be able to proceed. It is best to repeat this test using a few other multicast groups made up of random numbers to make sure that they all forward data as expected. We haven't been able to determine which kind of switch configuration produces this peculiar effect so we had to exclude these testbeds from our experiment.

While running the above code, it is also useful to monitor the interface on both nodes and a third node connected to the same LAN but not running either "mcastread" or "mcastsend". Because of the absence of listeners on the third node, the expected result is that multicast packets leave node1, arrive at node2, but are not visible on node3: this is the effect of MLD snooping on the switch, forwarding packets only to nodes which require them, and it is the best condition to run the experiment, as it produces the most useful results.

If the packets arrive at all nodes, the experiment can still proceed, but the network usage results for the multicast experiment will be less accurate. If possible the experiment should be moved to a different testbed.

In summary, we need to warn anybody wishing to repeat this experiment to first make sure that the system they are using is configured for IPv6 multicast, to avoid wasting their time on an experiment which will produce no results.

```
node1$ yes 'multicast testing' | \
       mcastsend -i $INTERFACE ff1e::42 4242
node2$ mcastread $INTERFACE ff1e::42 4242 > /tmp/datafile
node2$ (kill mcastread after a few seconds)
node2$ wc /tmp/datafile
 1430130  2860261 25742336 /tmp/datafile
node1$ (kill mcastsend)

node1$ yes 'multicast testing' | \
       mcastsend -i $INTERFACE ff1e::42:1234 4242
node2$ mcastread $INTERFACE ff1e::42:1234 4242 > /tmp/datafile
node2$ (kill mcastread after some time)
node2$ wc /tmp/datafile
 0 0 0 /tmp/datafile
node1$ (kill mcastsend)
```

Figure 9: Testing proper switch configuration (also see text)

References

[1] Quinn, B, and Almeroth, K. (2001). *IP Multicast Applications: Challenges and Solutions*. RFC 3170.
https://www.rfc-editor.org/in-notes/rfc3170.txt

[2] Sheffield, B. (2020). *IoT Updates with IPv6 Multicast - Updating a Billion Nodes from One Tiny Server*.
https://archive.fosdem.org/2020/schedule/event/iotmulticast

[3] Vida, R, and Costa L, eds. (2004). *Multicast Listener Discovery Version 2 (MLDv2) for IPv6*. RFC 3810.
https://www.rfc-editor.org/in-notes/rfc3810.txt

[4] Sheffield, B. *Multicast IoT Update Client and Server*.
https://github.com/librestack/iotupd

[5] Calvelli, C, and Sheffield, B. *Unicast (TCP/UDP) file syncing test program*.
https://github.com/librestack/unisync

[6] Sheffield, B. *Librecast IPv6 Multicast Router*.
https://github.com/librestack/lcroute

[7] Calvelli, C, Payne, E, and Sheffield, B. (2022). *Librecast: IoT Software Updates over IPv6 Multicast Archive of all experimental data [Data set]*. Zenodo.
https://doi.org/10.5281/zenodo.6382741

[8] Calvelli, C, Payne, E, and Sheffield, B. (2022). *Fed4fire experiment setup scripts*.
https://github.com/librestack/fed4fire

[9] *jfed 5.9 documentation*.
https://doc.ilabt.imec.be/jfed-documentation-5.9/index.html

[10] Calvelli, C. *lwmon: A **l**ight-**w**eight system **mon**itoring tool*.
https://github.com/librestack/lwmon

[11] *F0rth/mcast-tools: package containing IPv6-multicast routing daemons and tools*.
https://github.com/F0rth/mcast-tools