SystemImager and BitTorrent: a peer-to-peer approach for Large-Scale OS Deployment

Brian Elliott Finley (finley@anl.gov)
Erich Focht (efocht@hpce.nec.com)
Bernard Li (bernard@vanhpc.org)
Andrea Righi (a.righi@cineca.it)

This paper describes the integration of BitTorrent with SystemImager, as a transport protocol, for the deployment of OS images to large numbers of clients. Traditionally, the classical client-server model does not scale well, thus large scale deployments using SystemImager with its default transport protocol need to be staged. An alternative scalable and reliable transport protocol needs to be implemented and BitTorrent seems to be a natural fit.

Introduction

A brief overview of SystemImager

SystemImager is software that automates Linux installs, software distribution, and production deployment [7].

One key feature of SystemImager is that it is *distribution-agnostic* and is able to support heterogeneous hardware. This allows the deployment of any kind of *Linux* distribution (standard or even customized) to any kind of target machine. The main goal of the project is to make deployment of large numbers of computers easy. Typical environments include computer labs and render farms, but it has proven particularly popular in clustered computing environments, such as grid and high performance computing.

Another design feature that facilitates Linux distribution and hardware independence is that SystemImager works with file based (rather than block based) system images. An *image* is stored as a directory hierarchy of files representing a comprehensive snapshot of a machine, containing all the files and directories from the root of that machine's file system. Images can be acquired in multiple ways, including retrieval from a sample system (golden client), or by direct generation on the SystemImager server using third-party tools[1].

---

1 Tools can be distribution-independent, like *SystemInstaller* (http://systeminstaller.sf.net), or provided by the Linux distribution: like *YaST* for *SuSE*, *Debootstrap* for *Debian*, yum for Red Hat/Fedora, etc.

The standard method of image creation involves cloning of a pre-installed machine, the *golden-client*. In this way, the user can customize and tweak the golden-client's configuration according to his needs, verify it's proper operation, and be assured that the image, once deployed, will behave in the same way as the golden-client. Incremental updates are possible by syncing an updated golden-client to the image, then syncing that image to deployed machines using the *`si_updateclient`* command.

Images are hosted in a central repository on a server, called the *image-server*, and they can be distributed among the clients using different transports: *rsync* (the default), *multicast* (via Flamethrower[1]), *SSL encrypted rsync* (using a SSH tunnel), and now via *BitTorrent*.

BitTorrent

BitTorrent[2] is a peer-to-peer file distribution protocol that minimizes the burden on a server by distributing bits and pieces to clients (*peers*) in a swarm such that they can in turn distribute the pieces they have to others who need them. Developed by Brahm Cohen and written in the python language, this protocol is in wide use on the Internet for distributing large files. It is especially popular for distributing Linux ISOs and computer game demos because it is quite good at handling the huge spikes in the number of people wanting the files when they are first released. This protocol lowers the bandwidth costs of the provider and offloads costs and resources to the users wanting the content.

BitTorrent works by first having the content provider create a metafile (called the torrent file) which contains checksum/hash information regarding the file(s) as well as the location (URL) of the tracker which keeps track of connecting peers and file transfer progress. Once the torrent file is created, the tracker is started at the specified location and the content provider starts uploading the content in a process known as “*seeding*”. When content seekers connect to the tracker, they will download bits and pieces of the file from the seeder.

Peers start uploading the incoming bits to other peers as soon as they receive them and eventually switch to “*seeding*” mode when they have successfully received the entire file. Peers who have not yet achieved “seeder” status are known as leechers.

While not optimized for distributing files on a LAN, BitTorrent was still proven to provide much better performance and reliability than the rsync and multicast transports when used with larger numbers of clients. Support for multicast is provided via the Flamethrower program, however multicast[8] is not always stable and switch hardware from multiple vendors does not always work well together.

Therefore, a TCP based network protocol that can scale well is a more attractive solution, especially in complex heterogeneous infrastructures, like big server farms or Grid Computing[4] environments.

---

1 http://freshmeat.net/projects/flamethrower/

Comparison of the SystemImager protocols

The most frequently used protocol with SystemImager is *rsync*[9], thanks to its flexibility and the provided mechanisms of remote file synchronization. The design of this protocol respects the classical client-server approach, so it has a scalability problem: each client's bandwidth can be evaluated as $\frac{U_s}{N}$, where U_s is the upload bandwidth of the image server and N is the total number of the clients that are imaging at the same time. U_s is limited by the server's network bandwidth as well as the file system read performance. The drawbacks of this protocol are the performance limitations and the reliability in massive installations. A large number of clients imaged simultaneously can saturate the bandwidth of the image server resulting in the failure of a part or the whole installation. However *rsync* remains the best choice in installations with a small number of clients (< 15), due to its simplicity and very low overhead of the protocol, as compared to the other SystemImager transports.

Another approach is to use UDP over IP-multicast. A server-driven multicast strategy (integrated in Flamethrower[1]) theoretically scales very well compared to the classical client-server approach. In this case the image server pushes out the image only once and all clients belonging to the multicast group receive each UDP packet at the same time. In this case the download bandwidth of each client is equal to U_S. However, UDP over IP-multicast requires a highly reliable network. In fact, if a client misses a piece of the image for any reason, it might be forced to wait until the rest of the image has been sent and join the next transmission round.

To reduce the probability of error due to packet loss and to increase robustness of the protocol, the multicast transport uses an FEC[2] approach to transmit redundant data to the clients in a unidirectional mode. With FEC, clients need not acknowledge receipt of packets, nor can the request re-transmission of certain packets.

Each slice of data is divided in interleave stripes and for each stripe redundancy FEC packets are added. Striping is a way to increase performance, but since redundant data must be transmitted the theoretical bandwidth U_S is not fully exploited. In a typical configuration FEC uses 8 streams of redundant data. In this case the expected download rate of the clients is limited to $\frac{U_s}{8}$ [3].

A peer-to-peer protocol (like *BitTorrent*) allows each client to join the distribution of the image at any time, requesting the chunks it needs in any order, and leave the process when it can no longer contribute to the entire distribution.

---

1 http://freshmeat.net/projects/flamethrower/

2 Forward Error Correction: the sender adds redundant data to its messages, which allows the receiver to detect and correct errors without the need to ask the sender for additional data (http://en.wikipedia.org/wiki/Forward_error_correction).

3 Without FEC experimental results demonstrated that with a typical image of 2.5GB the 8-10% of the clients miss at least one packet.

With this approach the bandwidth of the image server U_s is a bottleneck only during the start-up phase of the imaging (when the clients have not yet received any piece of the image)[6].

Suppose that *x(t)* is the number of downloaders at a certain time *t, and y(t)* is the number of clients in seed-only state; we define the steady-state of *BitTorrent* when $\frac{dx(t)}{dt}=0$ and $\frac{dy(t)}{dt}=0$. Moreover we define *N* as the total amount of imaging clients.

In steady-state of a homogeneous environment the download rate of each client is not limited by U_s. Using the simple fluid model of Qiu and Srikant [3] the total upload rate is given by: $\mu(\eta x(t)+y(t))$, with η is the effectiveness of the file sharing and μ is the upload bandwidth of a generic peer.

If $\bar{x}$ and $\bar{y}$ are the steady-state values of *x(t)* and *y(t)* respectively, the aggregated bandwidth can be evaluated as: $\mu(\eta\bar{x}+\bar{y})$. Since there is only one seed (the image server), supposing a perfect effectiveness (η = 1) and supposing that all the clients are downloading at the same time, the theoretical limit of the upload bandwidth can be reformulated as: $\mu(N+1)$.

Hence the *BitTorrent* transport allows also to exceed the limit of U_s (obtained with *multicast*) when $\mu>\frac{U_s}{N+1}$. If clients and image server are homogeneous machines, μ = U_s and the upper bound can be defined as $U_s(N+1)$. In this case the available download bandwidth for a client is $U_s\frac{(N+1)}{N}$, but, obviously, in a homogeneous environment with a symmetric bandwidth the value is limited to U_s (the same value of multicast).

Moreover, due to the reliability of the BitTorrent protocol, the installation can successfully complete even under temporary network failure conditions. Reliability of each chunk (bit) is guaranteed by the BitTorrent protocol, since all pieces downloaded are checked using SHA1 hashes[10] .

Integrating BitTorrent in SystemImager

The main drawbacks of BitTorrent used as a SystemImager's transport are that:

1) the protocol is not designed to handle every element of a filesystem (special device files, symbolic links, file permissions, file ownership, timestamps),

2) it needs an auxiliary protocol to distribute torrents to the clients.

Point 1) can be resolved mapping the whole image on a single regular file. This is possible for example by tarring up the image directory prior to the file transfer, or to create the image filesystem on a regular file mounted in loopback. We opted for the tar approach, because in this way it is not necessary to pre-allocate the maximum space for each image (needed by the loopback file).

The problem 2) can be easily resolved using *rsync* to distribute the torrents to the clients. In fact torrents are very light compared to the images (more than an order of magnitude smaller) and with a typical image the scalability limits of *rsync* are not met.

To optimize the use of this transport the target is to reduce the *total installation time* (or deployment time). We define the total installation time as $T = \max\{t_i\},\ i = 1..N$, with t_i the time to deploy and execute the OS image on the client i.

The installation time for a generic client can be expressed in the form $t_i = \tau_i + K_i$ where τ_i is the time to download the image, that depends on the download bandwidth and K_i is constant[1]. The value τ_i is directly dependent on the download rate, that must be not limited by the total upload rate of the other peers: $U_{tot}(t) = \sum_{i=1}^{x(t)+y(t)} Us$.

The important condition is that $U_{tot}(t) \geq c \cdot x(t)$, where c is the maximum download rate of a client. It is essential to define an explicit condition to know when the contribution of a peer in the total upload bandwidth is no longer useful; in this case the peer can leave the swarm and reboot with the installed OS.

A simple solution could be to use the completion of the download as the exit condition for a client. Unfortunately this is not the optimal solution for all the cases: every client that is in the seed-only state can be a potential uploader for the other clients and leaving the swarm when the upload bandwidth is still used can introduce latencies and overheads in the installation of the other clients: clients must reconnect to the tracker to discover the new topology. Moreover, the seed-only state is particularly important when the clients are not homogeneous. In fact a client with a higher upload bandwidth is the best candidate to remain in the seed-only state after the download.

---

1 This is the time needed by the BIOS, the reboot, the driver detection, the disk partitioning, etc.

Choosing a BitTorrent implementation

Besides the original python implementation of BitTorrent, there have been various attempts to re-write/improve upon the original protocol namely in C, C++, Java, etc. Since we are looking for performance, we decided to search for C implementations of BitTorrent; a C application can also be easily compiled and included in either the initial RAM disk for SystemImager deployment or the BOEL[1] binaries. The original python implementation of BitTorrent can also be integrated with SystemImager, but in this case the python interpreter must be shipped with the client installation package or the scripts must be converted into executables.

Initial tests proved that the standard client was the best choice in terms of performance, compared to the other clients, so the natural choice was to include this implementation in the BOEL installation package.

The first approach was to include the python interpreter into BOEL. This solution has the advantage to provide a very powerful tool (*python*) into the BOEL environment, but it has some critical disadvantages. The most important is memory consumption. The python interpreter, modules, libraries, etc. take a lot of space in the basic installation package, and when imaging a large number of clients the transfer of this package can constitute the main bottleneck. A better approach is to exploit the python feature to freeze the scripts into common executables[2]. In this way the space needed by the BitTorrent binaries can be significantly reduced[3], since only the required components and libraries are included in the BOEL binaries (instead of the complete python environment).

With this approach the BitTorrent scripts can be considered as standard ELF[4] executables and they can be easily shipped into a initial RAM disk (*initrd*). This also makes it possible to use BitTorrent in the early stages of the boot process, reducing the bottleneck incurred by the use of common client-server protocols to transfer only a minimal startup environment and the BitTorrent binaries.

Thanks to the better scalability, the lower memory usage and the easy integration in the BOEL initial RAM disk the last approach was adopted.

The tool used to freeze the python scripts is *cx_Freeze*[5]. A generic “frozen” script is converted into a base executable which contains both the code of the script itself and the code of the required python modules[6].

---

1 BOEL (Brian's Own Embedded Linux) is the in memory OS environment used by SystemImager during an install.

2 http://www.py2exe.org/

3 The python environment increases the size of the basic installation package from 20 up to 30MB. The frozen scripts allow to reduce this limit to 5-6MB.

4 ELF (Executables and Linking Format) - http://en.wikipedia.org/wiki/ELF_file_format

5 http://starship.python.net/crew/atuining/cx_Freeze

Moreover, the needed libraries are placed into a destination directory and hardcoded paths[1] are replaced to obtain a full standalone application.

Overview of the installation process

The following is a schematic overview about the design of typical installation steps, both on the image server and on a generic client.

Deployment stages on the image server:

1. tracker start-up (the main component of the whole installation),
2. tarballs generation (the images to be deployed are tarred up and optionally gzipped),
3. torrents generation,
4. first seeder start-up (the first seeder runs many torrents distributions, in order to handle multiple images, BOEL binaries, overrides, etc. with a single seeder process[2]).

Deployment stages on the client side:

1. torrent files are downloaded using an alternative protocol (*rsync* or via *scp*),
2. BOEL binaries are distributed via BitTorrent,
3. image deployment via BitTorrent,
4. overrides deployment via BitTorrent.

Staging the image tarball

Unlike the multicast Flamethrower approach, BitTorrent cannot take advantage of "*tarpipe*"[3] for the image deployment. As a result, a file staging area is necessary to temporary store the full tarball prior to extraction into the client's file system.

The optimal solution in terms of performance is to stage the tarball in memory if the clients are equipped with sufficient RAM. In this case there is no overhead of the disk I/O while downloading the tarball, so the entire download process is not limited by the potential poor I/O performance of a single peer.

However, it is quite usual to have an OS image to exceed Gigabytes of data. Therefore it is necessary to identify a staging pool on the client file system.

---

6 Needed modules are identified by the "*import*" statements inside the scripts.

1 Dynamic library load path (also known as "rpath" and "runpath").

2 This is done via the command `btlaunchmany` (or `launchmany-console` in newer releases).

3 The multicast deployment can be seen as a stream of data, since the transmission of the packets respects a precise order. With BitTorrent there is not the concept of stream; every chunks are transmitted out-of-order, so this is not possible to parallelize the tarball distribution and its extraction through a pipe.

A simple auto-detection routine is used to find a proper pool comparing the size of the tarball and the size of the available space of the mounted partitions in the clients[1]. The preferred pool is always the “RAM disk” of the client (via *tmpfs*), followed by the */tmp* disk partition.

After the transfer, the tarball is extracted in the root of the client file system. The process of extraction is entirely asynchronous among the clients and during extraction they can continue to seed the image and the other tarballs.

Experimental results

Deployment time

To compare the speed and scalability of different transports we ran a number of tests in a real environment, installing subsets of the *BCX cluster* at CINECA[2]. The subsets, ranging in number from 1 to 150 nodes, always constitute a homogeneous environment.

Heuristically we found that 15 is the critical number of clients to choose *rsync* over *BitTorrent*. Under this value *rsync* is always the best choice. *Multicast*, on the other hand, works perfectly in terms of scalability, but the overhead of the protocol introduces a noticeable delay with this number of clients. Moreover the time of installation with *multicast* presents a remarkable variance from an experiment to another and, as expected, the speed does not depend on the total number of clients.

Looking at the results we found that 4.5min is the mean value of the time to install 15 clients, both with *rsync* and *BitTorrent* transports. The results were always the same in all the instances of this experiment (the time from an installation to another differs in terms of < 10 sec).

The mean value with *multicast* is 16min, but this value is quite dispersive (the time of installation ranges from 10 to 22 min on different instances of the same experiment). This high variance is probably due to other traffic in the network, since the switches were not isolated during the experiments and with multicast small differences in the background noise seem to strongly influence the final results.

The second test was focused to analyze the performance with a big installation, using a subset of 50 clients. In this case results provide evidence that there is a greater gap in the time of installation between *BitTorrent* and *rsync* transports (Figure 1).

1 Plus a breathing room of 100MB. This value has been heuristically determined and it seems to be enough to cover a lot of possible conditions.

2 http://www.cineca.it/en (appendix A contains the details of the installation environment).

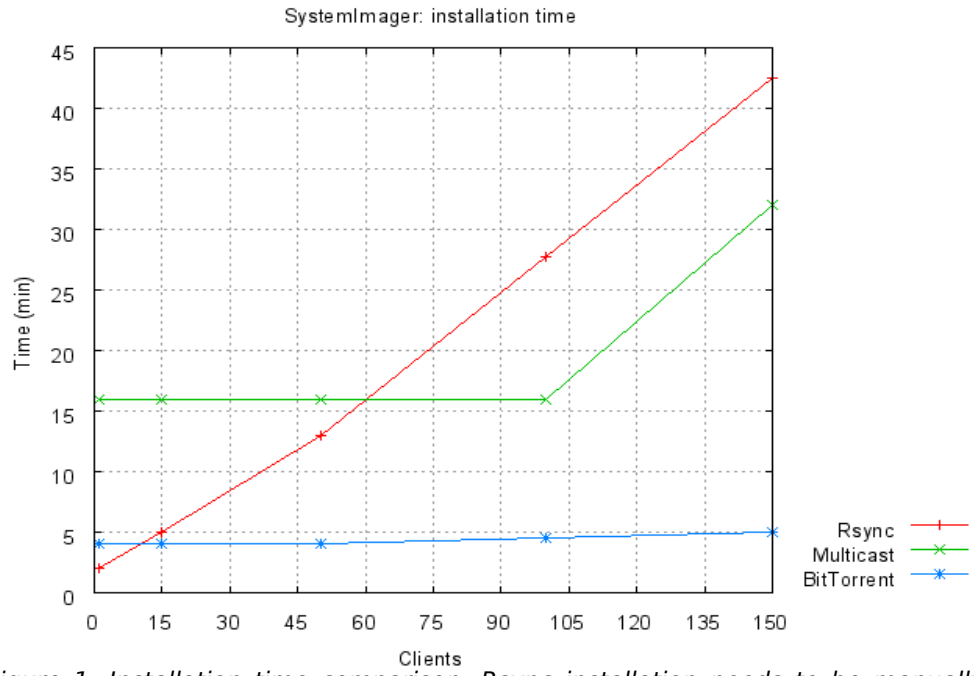

Figure 1: Installation time comparison. Rsync installation needs to be manually staged with more than 50 clients, so the installation time grows more than linear (there is a fixed time to boot the clients and prepare the installation). Multicast scalability is perfect theoretical, but with more than 100 clients the probability to miss a packet is very high. In this case clients need at least 2 sessions (the advantage respect to rsync is that this staging is done automatically by SystemImager and it is not manual). BitTorrent, instead, is very close the perfect scalability (horizontal line).

Rsync needed up to 13 minutes to complete the installation of all the clients and the image server reached its limits in terms of load and bandwidth usage. With more than 50 clients the image server starts to swap and some clients can fail the installation. This value can be considered an upper bound of the *rsync* transport and above this limit the installation must be split in different stages (this value is totally dependent on the CPU/RAM of server, and also the speed of the network adapter).

BitTorrent, instead, was perfectly scalable: it needed the same time (4-5min) to install all the 50 clients. Also the *multicast* transport obtained perfect scalability, but in terms of performance it was consistently inferior compared to *BitTorrent*. Since *rsync* is not usable to install more than 50 clients at the same time and since *multicast* performance is constant even with increasingly more clients, the last experiments were focused only to explore the limits of *BitTorrent.* Anyway, repeating the installation with 100 clients needed the same time: 4-5min to install the whole cluster. Only with 150 clients did the total time of installation increased slightly to 6min.

With *multicast* and the above mentioned number of clients, the probability for a node to miss at least one packet begins to be significant. It is not rare that one or more multicast sessions must be re-instantiated to send the image again to the faulty clients, doubling or tripling the total time of installation. This supports the requirement to have a clean, isolated and homogeneous environment for use with *multicast.* On the other hand, the upper bound of *BitTorrent* can be found only by saturating the bandwidth of the switches in the network.

Only in the last experiment does the weakness of BitTorrent emerged: each client is a potential uploader for the others at any time. When a client leaves

the *swarm*, after the complete installation of the image, it automatically closes the connections to the other *leechers*. This adds some latencies on the remaining nodes, because they need to reconnect to the tracker, ask for an updated list of the available peers, and instantiate new connections to continue the download. In huge installations (experimentally with more than 100 nodes) some *peers* can be affected by this latencies and the total installation time can be delayed. In this case performance can be improved allowing each client to leave the swarm only when it is no more useful for the other peers.

This behaviour can be heuristically modelled by adding a *seeding-only* wait time and stopping the *BitTorrent* client only when the upload rate reaches a minimum threshold[1]. Below this value the peer is considered free to leave the swarm and it can reboot with the freshly installed image. This adds a distributed delay on each single installation, but in huge environments the total time to install the whole cluster can be strongly improved.

Bandwidth analysis

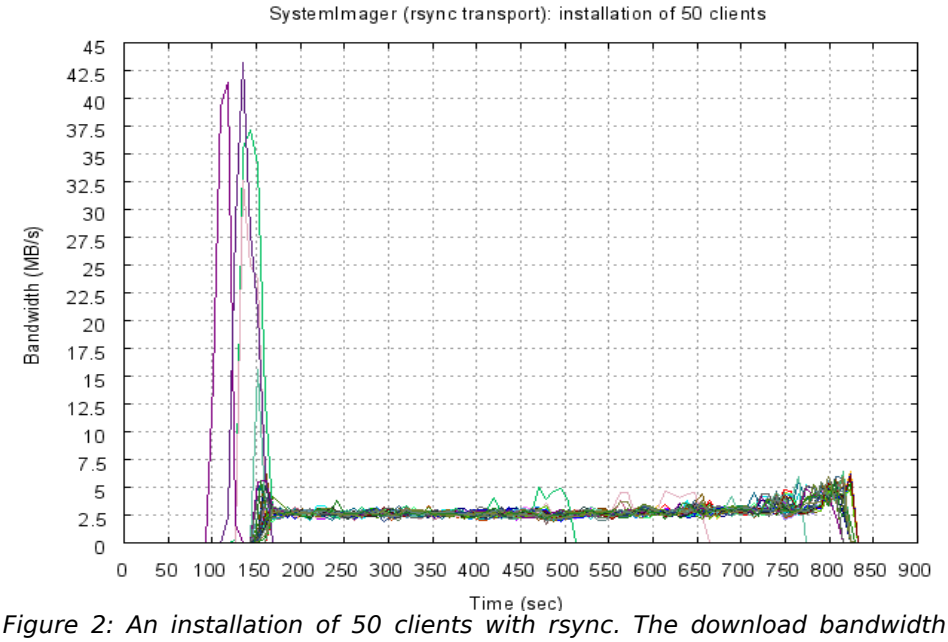

Figure 2: An installation of 50 clients with rsync. The download bandwidth of each client is inversely proportional to the total number of clients that are downloading the image at the same time. The upload bandwidth of the image server is the bottleneck of the whole process.

The monitoring features of *SystemImager*[2] allow to have a close approximation of the real download rate on each client during the imaging process.

The global bandwidth utilization can be considered the optimal parameter to evaluate the superiority of a protocol, because it is strictly dependent upon the

1 This can be done using the installation parameter BITTORRENT_SEED_WAIT=y (http://wiki.systemimager.org/index.php/Installation_Parameters). On a gigabit network we found heuristically the optimal value of 500KB for the minimum threshold.

2 http://wiki.systemimager.org/index.php/Monitoring

throughput and the total installation time.

As we can see in Figure 2 the download bandwidth on each client with *rsync* is inversely proportional to the total number of nodes that are imaging simultaneously. At the beginning and at the end of the transfer only few clients join the imaging group and the bandwidth reaches its maximum value. In the middle of the transfer the download rate is constant and stable at the lower values.

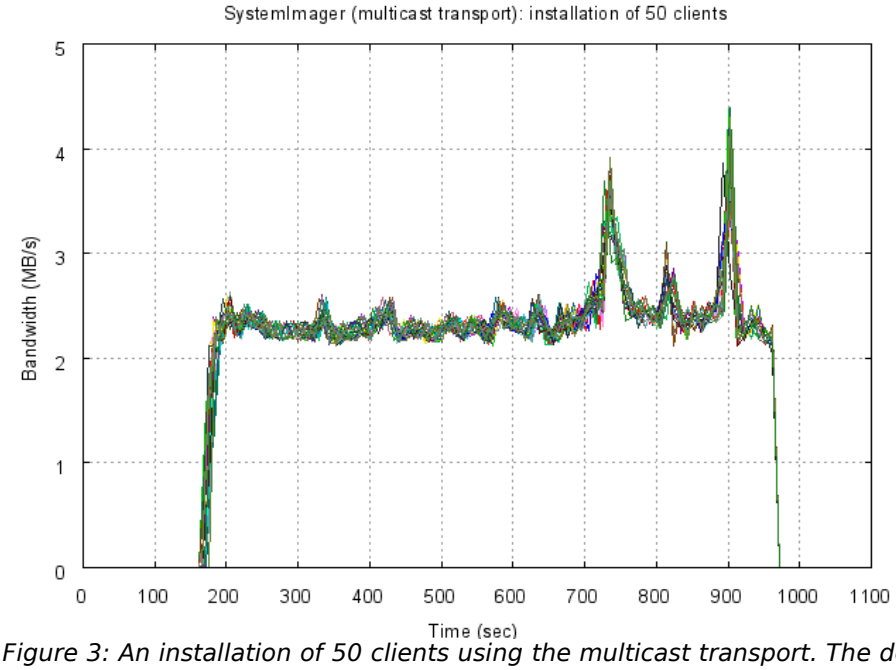

Figure 3: An installation of 50 clients using the multicast transport. The download rate is constant for all the clients at a generic time.

With multicast (Figure 3) the download rate is constant at a generic time for all the clients[1]. The rate is not perfectly constant in the time because of the overhead (or the optimizations) of the tarpipe process, the load on the image server and the *udp-send/udp-receive* processes: these can change the rate of the packets sent. Moreover, the multicast transport cannot be considered a *time-invariant* system[2] and the bandwidth graph presents differences also between instances of the same experiment.

1 The small differences in the download rates between the clients are only noise in the measurement process.

2 http://en.wikipedia.org/wiki/Time-invariant_system

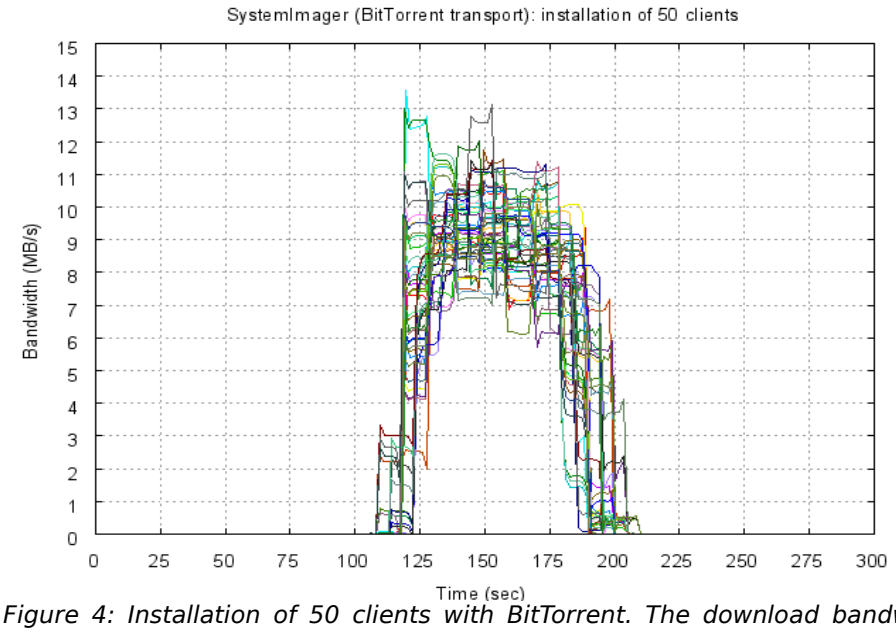

Figure 4: Installation of 50 clients with BitTorrent. The download bandwidth is limited by the upload bandwidth of the image server in the start-up (when the image server is the only seeder) and in the final phase (when the peers complete the installation and begin to leave the swarm). In the middle of the installation the bandwidth usage is greatly improved respect to the other protocols.

With *BitTorrent* (Figure 4) the pattern is the same in all the experiments: at the beginning of the transfer the aggregated download bandwidth of all the peers is limited by the upload bandwidth of the single seed on the image server. First blocks are randomly distributed among the clients and some blocks may be transferred before starting to exploit the other peers' bandwidth.

Only in this start-up phase and at the end, when the successfully installed peers leave the installation, the upload bandwidth of the image server is the bottleneck of the installation (Figure 5). Between these two phases each peer can exploit the upload bandwidth of the others, so the only limitation is due to the saturation of the network and the load on the image server is strongly reduced respect to the other transports.

Conclusion

The research of an alternative protocol capable of exceeding the limits of the classic client-server models for the installation of big clusters or complex grid-computing environments led to the exploration of peer-to-peer protocols.

The project characteristics and the adaptability of BitTorrent in the most heterogeneously interconnected environments constituted the motivation for the analysis, the implementation and the experiments with a new transport based on it. The experiments in a real production environment have put in evidence the performance advantages of BitTorrent in massive installations, compared to the other transports (*rsync* and *multicast*).

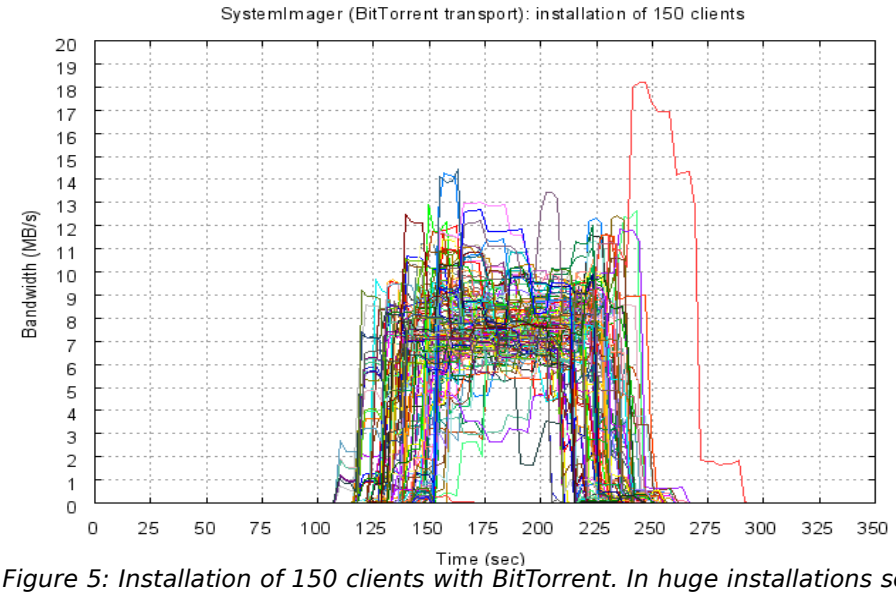

Figure 5: Installation of 150 clients with BitTorrent. In huge installations some clients may be affected by a delay caused by the peers that leave the swarm at the end of the transfer.

Also in terms of reliability the *BitTorrent-based* transport has showed interesting results. In particular with high load conditions of the network, result that other protocols (in particular reference to *rsync*) have not been able to guarantee exceeding critical thresholds in the dimension of the number of nodes installed at the same time.

Future work

BitTorrent has proven to be a viable protocol for rapid deployment of OS images to client nodes without substantial load on the image server. The speed of the deployment and the support for heterogeneous hardware provides the means to deploy virtual clusters which are built on the same resource pool, where it is possible to dynamically re-initialize workspaces on user demand.

The current work was done without any modifications of the protocol. However, further investigation can be taken to optimize the code for better performance in a controlled local network environment. The BitTorrent client can be further enhanced to support special files and also to carry meta-data regarding each individual file – e.g. timestamps, ownership, permissions, etc. This removes the overhead of creating a tarball of the image directory every time the image is modified.

So far we have only discussed using BitTorrent for the initial deployment of the image. It is also interesting to investigate how we can use BitTorrent to facilitate the massive update of clients (pushing changes/differences); this research can then be used to design a general-purpose file distribution/synchronization mechanism based on BitTorrent. The current experiments focus on creating differential tarballs containing the updates. This approach opens a path to image version management.

The current BitTorrent algorithm still has a client-server component: the tracker is still a single point of failure – when the tracker goes down, peers can lose contact with other and the file transfer can be halted. Work is currently being done to "*de-centralize*" BitTorrent in newer protocols such as *eXeem*. These new protocols will be analyzed for their suitability for integration with SystemImager as another file transfer transport.

Additional work must be done to enhance the security in terms of authentication, authorization and encryption to transfer data from the image server to the clients over insecure networks.

The BitTorrent protocol does not offer native security functionalities, so interesting improvements can be made implementing authentication mechanism to download the torrents and encrypting the tarballs before distributing them to the clients.

The *p2p* approach can be exploited also at the image server side to create distributed and redundant repositories of images. Multiple image servers can contain their own images and publish them via BitTorrent using a central tracker. The image tracker will forward the clients to the opportune image servers where the requested images are available. Having this grid of image servers would be the key ingredient to realize huge, scalable and reliable repositories of images.

References

[1] http://wiki.systemimager.org

[2] B. Cohen, Incentives Build Robustness in BitTorrent, 2003

[3] D. Qiu and R. Srikant, Modeling and performance analysis of BitTorrent-like peer-to-peer networks, 2004

[4] I. Foster and C. Kesselmann, The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufman Publishers Inc, 1998

[5] A.R. Bharambe, C. Herley, V.N. Padmanabhan, Analyzing and Improving BitTorrent Performance, 2005

[6] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, X. Zhang, Measurements, Analysis, and Modeling of BitTorrent-like Systems, 2005

[7] S. Dague, System Installation Suite – Massive Installation for Linux, 2002

[8] B. Quinn, K. Almeroth, RFC 3170: IP Multicast Applications: Challenges and Solutions, 2001

[9] A. Tridgell, The Rsync Algorithm, 2000

[10] D. Eastlake, P. Jones, US Secure Hash Algorithm 1 (SHA1), 2001

Appendix A:
Details of the installation environment

Following the technical information about the installation environment used for the experiments:

Image server

Model: IBM *e326m*
RAM: 4GB
CPU: 2 AMD Opteron(tm) Processor 252 2.6GHz
Network card: Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet
OS: RHEL4 U3 (x86_64)

Clients (from 15 to 150)

Model: IBM BladeCenter LS21
RAM: 8GB
CPU: 2 Dual-Core AMD Opteron(tm) Processor 2216HE 2.4GHz
Network card: Broadcom Corporation NetXtreme II BCM5706S Gigabit Ethernet

Network

IBM BladeCenter LS21 switches + 2 external CISCO Catalyst 6513 switches

Image: RHEL4 U3 (x86_64)

Size of the image: 2.1GB
Size of the gzipped tarball of the image: 606MB