Ephemeral State Assisted Discovery of Peer-to-peer Networks

Sylvain Martin and Guy Leduc *Member, IEEE*

***Abstract*—In the past few years there has been an impressive number of proposals for application-specific overlays or peer-to-peer networks. However the procedure to bootstrap peers in those networks has remained an under-researched topic, typically left as an implementation detail.**

In this paper, we first study the performance of history lists, a common decentralised bootstrap mechanism used in P2P applications. We then investigate how some lightweight network support, such as Ephemeral State, could help build a P2P application that would not depend on any centralised mechanism to discover other peers.

***Index Terms*—computers network, peer-to-peer computing, Internet, bootstrapping, ephemeral state**

I. Introduction

NOT so long ago, application designers were naturally thinking of how much would cost the server to support the number of clients they were envisioning. Deploying an impressive weaponry of load balancers and replicating systems, the monolithic servers have then turned into distributed systems that can cope with higher loads at lower cost. On the other side of the connection, however, the clients have remained blissfully ignorant of that process until the uprise of peer-to-peer (P2P) systems.

Nowadays, structured peer-to-peer[14] variants, and Distributed Hash Tables (DHT, [3], [4], [9]) in particular, seem to have unlimited applications and are seen in proposals for document preservation[5], multicast streaming [2] and even new network infrastructure [8] or routing [12], [13]. We can thus envision that DHT will greatly influence the design of new autonomic network architectures, e.g. as an integral part of name resolution or monitoring mechanisms. One major problem to be solved in these systems, before we can depend on them for network architecture, is how to bootstrap them. To the question "*How do you find a contact node in the overlay to join?*"[18], the answer is too often "*leave that to the end-user*".

In other contexts, that problem of communication *bootstrapping* is typically solved by a designated router on the local network advertising e.g. the local entry point to a distributed database (such as DNS server advertised through DHCP). But peer-to-peer networks are not part of the architecture and there is apparently no incentive for network operator to advertise the closest machine member of every possible P2P network – which may not even be in their own network.

Manuscript received on February 21, 2007; revised April 10, 2007. This work has been partially supported by the European IST-FET ANA project. Sylvain Martin is a Research Fellow of the Belgian National Fund for Scientific Research (FNRS).

S. Martin and G. Leduc are with University of Liège, Belgium.

In this work, we have studied the solutions proposed for P2P bootstrapping in existing implementations, aiming for a fully decentralised and self-configuring mechanism. As reported in section II, we mostly found a collection of hacks working around the lack of a proper support from the Internet architecture: ultimately, either the user will have to provide a contact node, or the software vendor will have to provision a server to process every bootstrap request.

We thus propose to use a lightweight extension of the router model – *Ephemeral State Store*[17] that offer time-limited storage of key/value pairs. In that extended architecture[16], special packets may use the publicly available storage on routers to advertise or look up for addresses of community members.

In section III, we propose a model of a commonly used bootstrapping mechanism (history lists) in which nodes try to join the community by contacting nodes that were their neighbours in the previous session. We also propose metrics to evaluate the performance of that mechanism under varying network conditions. We then show how the presence of "active" routers featuring an ephemeral store may improve the performance of that method in section IV.

We then give the reader an overview of how our proposal could bootstrap freshly installed nodes without requiring an initial list of peers in section V.

II. Joining a Peer-to-peer Network

In virtually all peer-to-peer applications, the operation of *joining the network* is separated into two steps:

1) find a *contact node* (or bootstrapping peer);
2) use that contact node to locate your neighbours in the network.

Depending on the desired network properties, the process of locating neighbours will of course vary, involving e.g. searching nodes with an identifier close to yours, detecting peers in your physical vicinity, etc. The incoming peer therefore needs a way to probe its current neighbourhood and ask peers for their neighbours list, in order to compare them and improve its own set.

The role of the *contact node* in this process is to provide the incoming peer an initial set of peers so that it can start this incremental neighbourhood selection. Most (if not all) architecture papers consider the location of that contact node as an implementation issue and assume the joining algorithm already has an "entry point" in the network.

There are different techniques implemented to locate that contact node, but none of them are fully satisfactory:

user knows: the application expects the user to provide the location of another running node to join, and provide a way for starting a 'stand alone' node. This is for instance the case in Chord[3].

static node: the application is bundled with a few addresses (or DNS names) of machines that will handle the bootstrap process. These machines (known as *pong servers* in the gnutella network[1]) will register every peer and reply with a list of peers that are believed to be still alive. The obvious drawback is that as soon as the pong server is put offline, the application simply stops working.

address-encoded: the user gives the application data that contain the address of the pong server to be used. This is for instance the case of the *BitTorrent* protocol[6] where a `.torrent` file contains both metadata of what you will download, and the location of the *tracker* that will be your "pong server". However, the `.torrent` file needs to be transmitted through some off-line means (e.g. using mails, newsgroups or a website) and there is no collaboration between peers downloading different contents.

pool service Jelasity et al. [20] suggest that a pool of potential peers should be maintained by a separate distributed service. While this allows quick spawning of P2P topologies from a set of stable nodes (interesting in the case of Grid computing), they still rely on off-line mechanisms to include the node into the pool in first place.

history file: rather than relying on some external pong server, each peer could store the list of neighbours it identified in the last session and try to reconnect to them (this is the case e.g. in FreeNet[7]). Depending on how long your system has spent offline, the size of that history and how dynamic the network is in general[11], this technique might offer fair performance or turn into a total nightmare. The actual success of the application will strongly depend on the presence of *super peers* that are running 24/7 at fixed IP address and on a mechanism to identify super peers in that history.

An intriguing alternative[18] suggests the use of a DHT (the Universal Ring) to associate identifiers of *service DHTs* with a list of contact nodes for that specific DHT. The authors advocate that the Universal Ring, being more widely supported that application-specific overlay, could be more easily located and could become part of the network architecture itself.

The techniques suggested to locate nodes of the Universal Ring are however not more convincing than what we found in other literature. Moreover, the proper operation of the Universal Ring requires that each member of the ring and each service obtain a digital certificate for their private key, which is – in our humble opinion – only practical in very restricted deployment scenarios.

III. History File Processing

Among those mechanisms, bootstrapping based on history files is the only one that is truly decentralised. This section presents a model of this process along with performance indicators that we will use through the rest of this paper.

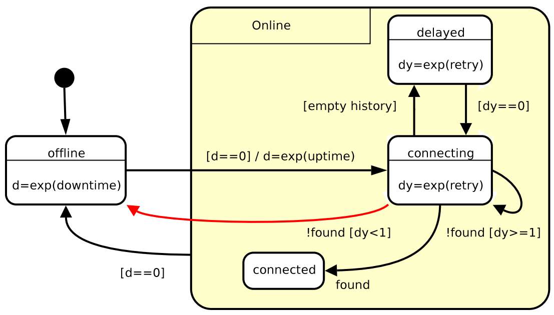


Figure 1. The UML state transition model of the peers

The reader should stay aware of the major inherent drawback of history processing: it requires an initial list. For machines that have been running at least one session, joining the network again is only a matter of patience, but for those system on which we just installed the P2P software, we need some out-of-band mechanisms to obtain a history. We will first assume that we have "imported" that initial list (e.g. shared by e-mail, from a friend inviting the user to join, or retrieved from a web search). We will see later in section V how we could bypass this requirement.

A. The Community Model

The community is made of a collection of *peers* that are connected to the Internet via a specific *stub* AS. Each peer belongs to a given *class* that defines its average online and offline time (e.g. "stud" nodes that are online for 2 out of 22 hours, "home" nodes that are online for 2 out of 14 hours and "desk" nodes that are online for 8 hours out of 24). We assumed here that, for a world-wide deployed system, we could reasonably consider that nightly shutdown have no globally observable effects.

Each peer's behaviour is defined by the state transition diagram of Fig. 1. When a peer enters or leaves the "offline" state, it will pick a random duration d before it leaves or returns to that state. As soon as the machine is powered up, it will contact peers in its history list ("connecting" state) observing a random pause dy between each attempt[1]. Since we want to avoid a partitioning of the whole peer-to-peer network, we only consider that a neighbour has been found when we manage to contact a node that is in "connected" state. When a peer has scanned its whole list without managing to contact anyone, it will enter the "delayed" state where it remains inactive for a short time interval before it tries scanning its list again.

Note that the system is greedy in the sense that a peer that has already found a neighbour will continue processing the rest of the list and the neighbour list of its neighbours. At shutdown, the peer will only keep a fixed amount of addresses in its history list (typically set to 10 in our experiments) and it will prefer long-established neighbours over other addresses.

[1] dy follows an exponential distribution around an average delay of 5 minutes, which we selected to approximate the timeout of a TCP connection establishment

Note too that the value of d is the *intended* session length. When the system is in the "connecting" state, we added a random return to "offline" modelling the behaviour of a frustrated user who connected his machine mainly with the idea of using the peer-to-peer system and just powers it off because the service is too long to set up.

In the following text, we will use the term "online" to refer to peers that are in one of "connecting", "connected" or "delayed" state.

B. Bootstrap Quality Indicators

In a typical simulation of the community we described, we start with a predefined amount N_0 of online peers. After a progression phase, the system will oscillate around the "equilibrium" amount of online peers on_{th} which can be derived from the online and offline time. F.i. when only *desk* (A) and *home* (B) classes are involved, we will have

$$\frac{on_{th}}{N} = \alpha \frac{u_A}{u_A + d_A} + (1-\alpha)\frac{u_B}{u_B + d_B} \tag{1}$$

where N is the total amount of peers and α is the ratio of *desk* nodes among them (u_A and d_A being respectively the average time spent online and offline in minutes).

In order to compare the quality of different bootstrap mechanisms, we measure the following indicators:

failed attempts: this is the ratio between the *unsuccessful* attempts and the total amount of connection attempts, that is, those who contact a machine that is down, that is not connected to the community yet, or an identifier that is no longer (or not yet) associated with a machine that can join the community.

frustration ratio: is the ratio between the number of sessions aborted before their scheduled "ontime" expires (e.g. due to a bored user) and the total amount of sessions in the simulation.

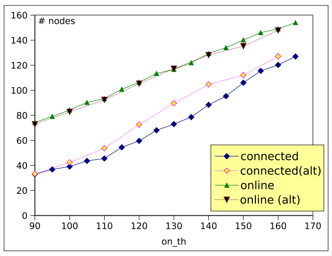
bootstrap efficiency: measures the percentage of the time spent online during which the node actually has access to the community.

$$eff = \frac{\sum T_{connected}}{\sum T_{online}}$$

For most indicators, the progression phase exhibits different values than the oscillation (equilibrium) phase. There are thus two kinds of scenario we might study:

bootstrapping: We initiate the community with a small amount of connected nodes which are all aware of one another (e.g. through coordinated configuration) and then study whether (and how fast) the whole community can reach the "equilibrium" phase. This allows us to simulate to what extent history-based peer bootstrapping is viable as the sole mechanism for a peer-to-peer community.

survival: We initiate the community with a number of connected nodes that is close to on_{th} and study how long this "equilibrium" can be maintained. This can be used to simulate the history-based peer bootstrapping as a fallback mechanism when another system (e.g. a pong server) suddenly becomes unavailable. Due to the lack of a global rendez-vous point for the community, the network can remain "online" only as long as at least two members remain *connected*. If we enter a state where the last *connected* member goes offline, any connection attempt will fail and the community will not recover.



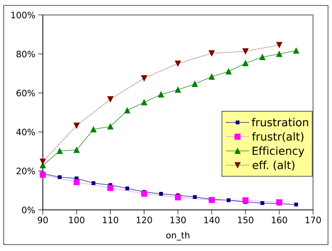


Figure 2. (left) Average number of online and connected peers for different values of the theoretical on_{th} parameter. (right) Efficiency and frustration ratio varying with the on_{th}

	u_A, d_A	u_B, d_B	α	on_{th}	len	eff
H+S	120,720	120,1200	0.387	110.1	94	0.43
D+S(alt)	480,960	120,1200	0.082	109.9	112	0.57
H+D	120,720	480,960	0.958	150.0	109	0.75
S+D(alt)	120,1200	480,960	0.750	150.7	151.4	0.81

Table I
COMPARING EFFICIENCY IN TWO SIMULATIONS WITH IDENTICAL on_{th}, BUT DIFFERENT MEAN SESSION LENGTH (len); α IS THE PORTION OF NODES OF TYPE A IN (1). UP AND DOWN TIMES GIVEN IN MINUTES.

C. Behaviour on a "Regular" Network

Intuitively, the probability of a successful contact will depend on the amount of connected peers, which itself depends on the amount of online peers. In turn, a high success probability will increase the number of connected peers and therefore reduce the probability of anticipated poweroff.

We ran a sequence of experiments with a 1000-peer community, varying the amount of "stud", "home" and "desk" machines to explore values of on_{th} ranging from 90 to 165 online nodes on average per simulation. The number of connected peers at the start of each simulation has been set to match on_{th} for that specific setup, allowing us to have more accurate results on a relatively short (2000 minutes) time span. We can see on Fig. 2, however, that the actual number of online peers may be up to 17% below the expected on_{th}, which can be explained by the fact that the formula for on_{th} doesn't take into account anticipated poweroffs.

As the theoretical number of online peers increases, we can see that the frustration ratio decrease from 19% to 3%, which is accompanied by a more accurate approximation of the actual average of online peers by on_{th}. A few additional experiments with a home:desk ratio of 5:5, 3:7 and 1:9 (leading to on_{th} values of 237, 275 and 313 respectively) confirmed the progression we observed. With an average of 313 online machines, the efficiency reaches 97% and the frustration ratio tends towards 0, which makes the relative error on on_{th} of only 2%.

We repeated the experiment using only machines from "stud" and "desk" class, and report the result in the "(alt)" data series of Fig. 2. While the average number of online peers for these simulations is virtually identical to the figures obtained with "stud+home" and "desk+home" mixing of the previous simulations, the alternate simulations exhibit a higher efficiency. We then investigated the mean session length (see table III-C), which revealed as expected a longer mean session in the alternate simulations. This mean that, for identical average community size, it is preferable to have fewer "better peers" if they have a longer average session length.

One should note that in both simulations, efficiency and frustration can be directly expressed as a linear function of the probability of a successful connection attempt. This confirms our a priori feeling that we should try to improve the probability of a successful "hello" if we want to improve the overall system performance.

IV. Active Domains boosting P2P

A simple way to increase the chances for a hello message to be successful is to make it more capable of detecting running peers on its way. With regular IP processing, a "hello" packet will test only one machine and will be successful only if that machine is connected[2]. If instead the "hello" packet could be distributed to all the machines running in a given domain (e.g. all the clients of one ISP), our chances of having a positive answer will become:

$$P(success) = 1 - (1 - P(conn))^N$$

P(success) probability of a successful reply if the packet is targeted to a machine in domain D.

P(conn) probability for a machine of D of being connected. We assume in this formula that the domain D is homogeneous and that all its clients have an equal probability of being member of the community.

N is the number of clients in domain D that have already been members of the community.

Such a technique, however, would lead to excessive unsolicited traffic. We can still achieve similar improvement with substantially less overhead if edge routers of the ISP domain are capable of storing information about which of its clients are member of the community. Such an ISP is then called an *active domain*, and the community members connected through it are said to be *active peers* if their software is capable of periodically sending active packets that store their address on the border router.

In the following simulations, we model an *active domain* as a domain on which it is possible to issue a *probe* that will return the address of a random peer in the community that is client of that domain. Even a successful probe that returns an address does not immediately cause a transition to the *connected* state. Rather, the address is pushed at the head of the 'pending list', and will be processed as the next peer to probe.

There are two situations where a peer can benefit from the active routers support: either it could have the address of an "active peer" in his history list, or it could be itself a client of an active domain and do a 'local probe' for connected peers.

[2]in that case, we simply have $P(success) = P(conn)$

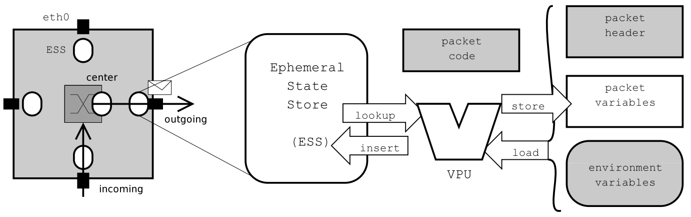


Figure 3. The WASP router, illustrating location of ESS/filter components around the routing core and allowed data movements in the virtual processor (VPU).

The results presented in the following sections assume that both probing mechanisms (i.e. local and remote) are in use. Our early simulations with remote probing only showed that adding local probes does improve efficiency, but only by a few percent.

Note that the above formula only holds for domains where $P(conn)$ of clients is independent. This won't be the case, for instance, if D is a geographically concentrated area and if all machines in D observe similar nightly shutdowns – such as in a corporate network[15]. It should be considered as a theoretical upper bound on $P(success)$ rather than a way to predict it. Note too, that it is the probability of a successful connection attempt *given that the packet is sent to an active domain*. We should thus moderate it with the probability that an entry in the history list is a client of an active domain.

A. WASP : Our Lightweight Active Framework

Compared to most of the active network frameworks proposed in the last years, the support we require from the network in this paper is extremely modest. The most elementary requirement is the presence of a publicly available *state store* on the active router where packets could resolve a community name (or its hash, the *community key* K) into an address (or a short list of addresses) of community members.

In our test-bed implementation, we have opted for an *ephemeral store* [17] associated with a filtering component at each network interface on the router. A packet that crosses the router can request processing only on the component attached with its *incoming* and *outgoing* interfaces (see Fig. 3). In the resulting platform (WASP: World-friendly Active packets for ephemeral State Processing [16]), the filter component evaluates whether packets received or transmitted through their associated interface contains a WASP bytecode program and interprets it as needed.

The program can mainly lookup or modify entries in the Ephemeral State Store (ESS) using keys included in the packet, write back computation result in the packet's "scratch" area and perform basic computations. It will also tell the router whether the packet should be dropped, forwarded, or returned anticipatively to its source – which will be useful in our case where the destination might just be powered off.

Using this framework, the peer discovery protocol could be implemented with two (simple) WASP programs:

`peer_adv(K, addr)` this packet is sent periodically by community members to a random peer. When processed on an outgoing interface, it will try to add its source address *addr* to the list maintained under key K in the

ESS. If the list is full, it will return to its source where it could trigger a self-regulation mechanism (see below).

peer_probe(K) this is the probe packet sent together with connection attempts to the addresses mentioned in a history list. When processed on an interface, it will look for a list of peer in the ESS and copy the addresses into packet's "scratch" area. It will return to its source if any address has been found and goes on its way otherwise.

A peer that bootstraps will first issue a *local probe* that looks for a WASP router with membership information in its local domain (e.g. using a random target address and a small TTL). Then, together with the connection establishment attempts for each address in the history list, the peer issues a *remote probe* that will come back if the community key is found in a router on the path to the probed address. The same `peer_probe()` program executed on the *outgoing* interface of the source's domain or on the *incoming* interface of the destination's domain can implement local or remote probe respectively.

On each WASP router, we only need one entry (32 bytes) per community, independently of the number of peers that are members of this community. Each entry should be refreshed at least once per "ephemeral period" τ (typically 10 seconds), and preferably by drawing a random delay uniformly between $\tau/2$ and τ to approach the random peer selection mechanism mentioned in the model.

While a single hardware context on a modest network processor is reported to handle about 200,000 ESS requests per second [19], thus potentially supporting up to 2 million members in a single domain like a charm, it is clear that it would be preferable to use the feedback information provided by returned `peer_adv` to estimate the amount of local peers and adapt advertisement period accordingly.

Similarly, if we have each online peer issuing one probe every 5 minutes on average, a single hardware context could handle aggregated requests for around 63 millions peers, in the unlikely event that they all probed an address behind our router. Still, even a simple network processor such as the IXP1200 used in [19] could use up to 16 hardware contexts for WASP packets processing, and we could easily offload the egress router by adding WASP processing on access routers too.

B. Keeping the Community Running

In these simulations, we will investigate the benefit of active networks during a "survival" scenario. 1000 nodes are first randomly assigned to the different domains which are then randomly "activated" to reach 100 active peers[3]. All the following simulations use a population of 950 "home" machines and 50 "desk" machines and are started with $on_{th} = 150$ connected peers. Each simulation set contains 20 independent runs of 2000 minutes.

regular This is our "reference" simulations set, with no active peers. As detailed in the previous section, this leads to an average of 106 connected peers, a system efficiency of 75% and a frustration ratio of 4%.

act:med This set has on average 103 active peers and an average domain size of 20 peers. It leads to an average of 128 connected nodes and improved efficiency of 87%. We can also notice the low frustration ratio of 0.99%

act:small In this set, we have on average 100 active peers and an average domain size of 10 peers, which leads to 125 connected nodes and system efficiency of 86%. Smaller domains thus clearly offer more modest performance boost over the "regular" network, but still, this remains clearly a boost over the reference set.

act:huge Here we have only one active domain whose size is on average 113 peers, resulting in an average of 130 connected nodes and system efficiency of 89%. It also has the lowest frustration ratio of 0.66%.

[3]As a side effect of this policy, we will experience a higher variance in simulations featuring only 10 domains

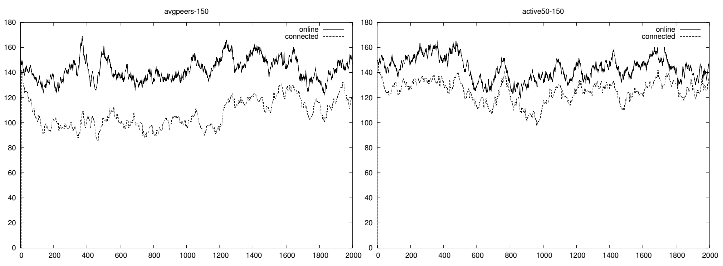


Figure 4. plotting the amount of online and connected peers over time in a regular (left) and an active (right) simulation

All the active simulations thus outperform the reference simulations, be it by the size of the community they manage to maintain, the number of anticipatively terminated sessions (e.g. frustrated users), or the amount of time required to get connected to the system. These performance gains still hold with a population of 2000 peers as long as other parameters are also scaled accordingly (e.g. 200 active peers and preserving the average domain size).

It should also be mentioned that, for the user starting his machine, the improvement of "only" 14% in efficiency in our simulations implies an application that is ready for use twice faster.

C. Getting the Community Running

Now that we know the average number of connected peers that the system is able to maintain in various settings of active domains, we can compare how fast the system reaches its "cruise level". The following simulations have been started with relatively few ($N_0 = 30$) connected machines (that could e.g. be the set of systems that has been up during the weekend), and let the system evolve to restore its equilibrium.

Table II summarises the result we obtain for those simulations. For each setting, it shows the average amount of connected peers the system can sustain (as measured in section IV-B) and the average time needed to reach that level for the first time (t_{sust}). The value of t_{sust} identifies a "knee" point on the curve where the system has reached its equilibrium and now oscillates around the average value. We also measured

	$sust$	$postavg$	t_{80}	t_{100}	t_{sust}
regular	106	108	607	815	943
act:small	125	129	263	348	676
act:med	128	131	196	265	562
act:huge	130	131	150	226	584

Table II
NUMBER OF CONNECTED PEERS AT EQUILIBRIUM ($sust$), AFTER EQUILIBRIUM LEVEL HAS BEEN REACHED ($postavg$), AND TIME (IN MINUTES) REQUIRED TO REACH 80, 100 AND $sust$ CONNECTED PEERS.

act. $\backslash P_{dyn}$	0%	10%	20%	30%	50%	70%
0%	72.4	67.9	64.7	58.3	33.3	20.3
5%	83.4	81.6	79.8	78.3	68.5	51.0
10%	86.3	85.6	85.1	84.7	80.8	74.7
20%	90.2	90.1	90.0	89.3	86.2	85.7

Table III
IMPACT OF DYNAMIC ADDRESSING ON SYSTEM EFFICIENCY FOR VARIOUS RATIO OF ACTIVE DOMAIN SUPPORT.

the actual average value *past* the knee point (that is, for t in $t_{sust} \ldots 2000$), as reported in the *postavg* column[4]. So not only the active routers can help having more nodes connected on Mondays morning, but it can also cut to 60% the time required to rebuild the community (t_{sust}) in the regular network. If we are rather interested in how fast each setting can reach an arbitrary value, we can see in columns t_{80} and t_{100} that even the less optimistic scenario (act:small, with an average of 10 members per active domain) is more than twice as fast as the regular network, and that with larger active domains, we can even be 4 times faster.

Another benefit brought by active networking here is the size of the initial set required to actually bring the community connected. We reproduced the experiment with ($N_0 =$) 25, 20, 15, 10 and 5 machines initially connected to see how many of the 20 simulations could still "take off" and grow to the expected equilibrium value. Indeed, if new peers cannot find those "initial members" quickly, the initial members themselves might disconnect and we will end with an "aborted" community where no one can connect anymore.

With a regular system, things starts getting wrong with $N_0 = 20$, where 10% of the simulations aborted, and further degrades so that with $N_0 = 12$, we have less than 50% chance of seeing the community taking off. On the other side, a system with 100 active peers in a configuration similar to *act:med* could still take off in all simulations with $N_0 = 12$ and gives 90% and 60% of chances of a successful take off with $N_0 = 10$ and $N_0 = 5$ respectively.

D. Other Affecting Parameters

The important random variable through these simulations is the probability of finding an online machine in a given time period T. The different scenarios we investigate in this section all alter the 'default' probability.

It is clear, too, that this probability depends on how many different addresses we can test during period T. The delay between two connection attempts, for instance, directly influences the percentage of time spent online, regardless of whether or not we have active nodes.

Note that, in most implementations, the peer will scan several addresses in parallel. There is however, a maximum amount of attempts that we could do on a given platform, meaning that e.g. we could have an average number of scanned nodes on a τ time slice that is k times higher than what we observe in our simulations. Simulation results can still be applied to such a system if we assume that the individual probability for a node to be online is actually k times lower in the real system than in the simulation (e.g. a given node doesn't connect 2 hours every day, but rather 2 hours every k days).

E. Dynamic Addressing vs. Active Domains

So far, we have assumed through all our tests that a peer leaving the system and then joining it again will always reuse the same address. However, an increasingly high number of ISPs only offer *dynamic* addressing to their clients: every time a machine connects to the Internet, it will receive one of the addresses from the ISP pool, that it will keep during its whole session, but chances are very slim that the same address is allocated twice in a row to the same machine, especially hours after the last session.

In the following simulations, peers have probability P_{dyn} of being client of a *dynamic domain*. Those domains will assign a new address to their clients every time they connect.

If we assume that the community members (both online and offline) represent a fraction k of the number of addresses available in the domain's pool, there is now a probability $(1-k).P_{dyn}$ that an address we find in our history list no longer corresponds to a peer, but that it rather has been reallocated to a machine that doesn't run the P2P software.

The first row in table III shows the efficiency of the P2P community with P_{dyn} varying from 0.1 to 0.7 and k being fixed to 0.1 (i.e. the size of each dynamic domain's pool is 10 times its number of peers[5]) without any active router. As we can see, the system efficiency quickly degrades when we add more dynamic domains. We also observed a significant degradation of the frustration ratio and the average community size.

On the other side, adding *active* networks results in an improvement of the bootstrap efficiency and other studied parameters. Moreover, it is not important to have a high number of active domains to obtain a significant effect: 5 percent of domains being active is enough to gain 11% of bootstrap efficiency, but we need half the domains to be active if we want to gain another 11% of efficiency.

It is interesting to note that a dynamic domain that is capable of storing information for active packets will behave here like a static, active domain. Indeed, the fact that new addresses are allocated every time a node connects is compensated by the

[4]We can observe here that postavg is systematically slightly above the average of section IV-B, which could be due to the shorter runs not being able to compensate for the oscillations amplitude

[5]While there are typically almost more clients than addresses in an ISP that does dynamic addressing, it would be utopian to assume all those clients already run our P2P software

fact we can obtain the address of a community member (if any) using any address previously seen in that domain.

Moreover, as depicted in Table III, the presence of a few active domains in the system can compensate the degradation resulting from the presence of dynamic domains. Even with only 10% of the domains supporting the active packets, we can almost annihilate that degradation and keep the same efficiency regardless of the amount of dynamic domains in the network.

There is a new phenomenon that appears with $P_{dyn} > 0.4$. Some of the nodes might end up with only unassigned addresses in their history list[6], meaning that they have virtually no chances of connecting to the peer-to-peer network. From $P_{dyn} = 0.7$, the phenomenon can no longer be neglected since it will affect on average 2% of the members – a value that will quickly grow over 25% of the members when $P_{dyn} = 0.9$.

Note that even in extreme conditions such as $P_{dyn} = 0.9$, where a regular network couldn't maintain the community alive for more than a couple of hours, the presence of 10% of active peers allows the system to survive for arbitrarily long time, although with a degraded efficiency (around 66%) and a significant number of nodes that might end up with a useless history list (18%, against 26% without active nodes).

Indeed, the community model only takes into account the age of a neighbour when it picks the addresses it will archive in its history list for the next run. While an active address has more chance to be kept from one run to the other, the address from an active disconnected peer will not be preferred over a dynamic, connected peer, although the latter will likely be useless in the next run.

V. Avoid the Need for an Initial List

So far, we have illustrated that exchanging member addresses at active routers could help the peer-to-peer community to recover from unusually low activity and that its efficiency can be boosted through the improved probability of a successful connection attempt.

Still, there is a major drawback of history-based peer-to-peer systems we haven't addressed yet: the need for an initial history list. An instance of the P2P software freshly installed on a machine has no other system to connect to. Most existing system will overcome this through off-line process to obtain this list, such as publishing it on a forum or manual transmission through e-mails. In this section, we will review a collection of techniques enabled by the presence of WASP in the network that could avoid that need.

First, when the new machine's ISP runs WASP routers, we might build our initial list by looking for other peers in our own domain. Picking a random destination address and sending a probe should be sufficient here to hit the border router on which other peers of the same domain have advertised their address. Relying only on this automatic setup might work well when the software is actually "pushed" by the ISP (such as a GRID platform promoted by the managers of the domain), but it will generally be frustrating for lambda users downloading some P2P software regardless of whether their ISP supports WASP or not and whether there is already a sufficient user base in their vicinity.

[6]the simulator considers an address unassigned when it belongs to the pool of a dynamic domain, but not currently assigned to any member of this domain

We can envision another successful deployment scenario if the software vendor can afford setting up a WASP router at the entry point of his own network domain. In that case, the software can be designed so that, as a last option, it probes the vendor's domain for some initial contact node. One might valuably argue, though, that this is no different from running a pong server from an architectural point of view.

The good thing WASP routers offer here is that all we need to find is *active domains* that contain members, not members themselves. If the vendor cannot upgrade to WASP routers but WASP is sufficiently spread on the network, he could simply run a scanning software that would probe all ISPs and include a list of active domains as the "initial peers list" for the shipped software.

If we want to build a peer-to-peer system based solely on history lists, as opposed to systems where that list is just the most scalable and preferable mechanism before we fall back to a pong server, it is clear that we need an additional way of rebuilding the list when it is empty or useless. Once again, thanks to the presence of WASP router, the process of scanning the network for other peers is greatly simplified by the fact that we simply have to send a packet to any address (even if not currently assigned to a running machine) of an active domain where another peer is running to get a match. Knowing that, we can opt for different techniques that will gather addresses to test:

netstat: The P2P application might periodically run a netstat-like tool to learn the current connections the hosting system has with other machines, then send probes to those locations to see whether peers can be found.

web browsing: Rather than waiting for the user to establish connections, we might import the history of previous connections from e.g. the user's browser. We can then build a list of destination addresses out of the URLs and check if we can see any active router.

address book: Another potential source of peer addresses would be the address book of the user's mailing application. Out of "John.Smith@BigISP.com", we can extract the IP block associated with "BigISP.com" and send active probes to see whether there are online peers in that location.

The drawback with the "web browsing" approach is that there are unfortunately little chance that popular web servers are co-located with potential peers. There is a way an active "server" site could be helpful if it is popular enough. Indeed, we could technically use a website like slashdot or google as a rendez-vous for machines looking for a peer to join (which might visit the site too). It would however require the community to pro-actively scan for active routers frequently visited by users and to maintain ephemeral state present by periodically refreshing on routers that would otherwise never carry traffic for our P2P activities.

Comparatively, the "address book" scheme is more likely to point us to domains that host *users* rather than services. With a simple component filtering out most frequent webmail providers, we can concentrate our search efforts on parts of the network that could be running compatible P2P software.

VI. Conclusion and Future Work

It is our belief that the currently existing peer-to-peer applications lack a scalable, robust and user-friendly way to let peers join the network, be it for the first or the nth time. We have however no doubt that peer-to-peer technology has now given enough proofs of its potential and that we are likely to see more and more applications and architectures involving P2P networks in the future.

Through this work, we have shown that small modifications inspired by active networking research would allow a significant improvement of P2P system performance, and that it might even allow us to distribute peer-to-peer software without having to dedicate online resource to support it.

As we detailed in our previous work[16], we already have a working prototype of our router extension for the Linux kernel. Basing on the code from the ESP project[19], we are currently busy porting it to the Intel IXP2400 network processor to demonstrate the feasibility of custom ephemeral state processing at wire speeds up to 1Gbps on each interface.

In section II, we mentioned the two-phased nature of joining a peer-to-peer system. The second phase (neighbours selection) is oversimplified in our simulations, and while we have good hope that our proposal could equally apply to e.g. a Chord ring, we still need additional simulations with an enhanced model taking into account ring maintenance algorithms to validate our belief.

Finally, the approaches for initial list avoidance presented in section V clearly need more research before getting convincing. We have however good faith that they can be useful building blocks of a heuristic technique which – given sufficient WASP support from the participating domains – could strongly reduce the number of cases where installing the software requires extra user intervention.

Acknowledgement

Thanks fly to Cyril Soldani and Cyril Briquet for their timely help.

References

[1] Gnutella protocol v0.4, 2001, http://www9.limewire.com/developer/gnutella protocol 0.4.pdf.

[2] M. Castro, P. Druschel et al. : *"SplitStream: High-bandwidth multicast in a cooperative environment"*, in Proc. of SOSP'03, New York, Oct. 2003, pp. 298 - 313.

[3] Stoica , Morris et al. : *"Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications"*, in Proc. of ACM SIGCOMM 2001, pp. 149-160.

[4] A. Rowstron and P. Druschel : *"Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems"*, in Proc. of Middleware 2001, LNCS 2218, pp. 329-350.

[5] Kubiatowicz J., et al. : *"Oceanstore: An architecture for global-scale persistent storage"*, in Proc. of ACM ASPLOS IX, Cambridge, USA, Nov. 2000, pp. 190-201.

[6] BitTorrent Protocol Specification *"http://www.bittorrent.org/protocol.html"*

[7] I. Clarke, O. Sandberg et al. *"Freenet: A Distributed Anonymous Information Storage and Retrieval System"*, in Proc. of Workshop on Design Issues in Anonymity and Unobservability, Berkeley, USA, July 25-26, 2000, LNCS 2009, pp. 46-66.

[8] I. Stoica, D. Adkins, S. Zhuang et al. : *"Internet indirection infrastructure"*, in Proc. of ACM SIGCOMM Computer Communication Review Volume 32 , Issue 4 (October 2002).

[9] P. Maymounkov and D. Mazieres : *"Kademlia: A peer-to-peer information system based on the XOR metric"*, in Proc. of IPTPS 2002, Cambridge, USA, March 7-8, 2002, Revised Papers, LNCS 2429, pp. 53-65.

[10] S. Sen and J. Wang : *"Analysing peer-to-peer traffic across large networks"*, in Proc. of 2nd ACM SIGCOMM Workshop on Internet measurment, Marseille, France, 2002, pp. 137-150

[11] S. Rhea, D. Geels, et al.: *"Handling Churn in a DHT"*, in Proc. of USENIX Technical Conference, June 2004, pp. 127-140.

[12] M. Caesar, M. Castro et al. : *"Virtual ring routing: network routing inspired by DHTs"*, in Proc. of ACM SIGCOMM, September 2006, Pisa, Italy, pp. 351 - 362.

[13] M. Caesar, T. Condie et al. : *"ROFL: Routing on Flat Labels"*, in Proc. of ACM SIGCOMM'06, Sept. 2006, Pisa, Italy, pp. 363 - 374.

[14] E. Keon, J. Crowcroft et al. : *"A Survey and Comparison of Peer-to-Peer Overlay Network Schemes"*, Communications Surveys & Tutorials, IEEE, 2005, pp. 72–93.

[15] W. Bolosky, J. Douceur et al. : *"Feasability of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs"*, in Proc. of SIGMETRICS 2000, Santa Clara, USA, pp. 34-43.

[16] S. Martin and G. Leduc : *"Interpreted Active Packets for Ephemeral State Processing Routers"*, in Proc. of IWAN, Sophia Antipolis, 2005, LNCS 4388.

[17] K. Calvert, J. Griffioen, and Su Wen : *"Lightweight network support for scalable end-to-end services"*, in Proc. of ACM SIGCOMM, 2002, pp. 265-278

[18] M. Castro, P. Druschel : *"One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks"*, in Proc. of SIGOPS European Workshop, Saint-Emilion, France, 2002, pp. 140-145.

[19] K. Calvert, J. Griffioen, N. Imam, J. Li : *"Challenges in Implementing an ESP Service"*, in Proc. of IWAN'03, Kyoto, LNCS 2982, pp. 3-19.

[20] M. Jelasity, A. Montresor and O. Babaoglu, *"The Bootstrapping Service"*, in Proc. of IEEE ICDCSW'06, Los Alamitos, CA, USA, July 2006, p. 11.