Reversible Phase Transitions in a Structured Overlay Network with Churn *

Ruma R. Paul[1,2], Peter Van Roy[1], and Vladimir Vlassov[2]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
{ruma.paul,peter.vanroy}@uclouvain.be
[2] KTH Royal Institute of Technology, Stockholm, Sweden
{rrpaul,vladv}@kth.se

Abstract. Distributed applications break down when the underlying system has too many node or communication failures. In this paper, we propose a general approach to building distributed applications that lets them survive hostile conditions such as these failures. We extend an existing *Structured Overlay Network (SON)* that hosts a transactional replicated key/value store to be *Reversible*, i.e., it is able to regain its original functionality as the environment hostility recedes. For this paper we consider the environment hostility to be measured by the *Churn* parameter, i.e., the rate of node turnover (nodes failing and being replaced by new correct nodes). In order to describe the qualitative behavior of the SON at high churn, we introduce the concept of *Phase* of the SON. All nodes in a phase exhibit the same qualitative properties, which are different for the nodes in different phases. We demonstrate the existence of *Phase Transitions* (i.e., a significant fraction of nodes changes phase) as churn varies and show that our concept of phase is analogous to the macroscopic phase of physical systems. We empirically identify the *Critical Points* (i.e., when there exists more than one phase simultaneously in significant fractions of the system) observed in our experiments. We propose an *API* to allow the application layer to be informed about the current phase of a node. We analyze how the application layer can use this knowledge for self-adaptation, self-optimization and achieve reversibility in the application-level semantics.

Keywords: Phase Transition · Maintenance Strategies · Churn

1 Introduction

A distributed application breaks down when there are too many node or communication failures, in which case the application can revert to an "offline mode"

* This research is partially funded by the SyncFree project in the European Union Seventh Framework Programme under Grant Agreement No. 609551 and by the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030. Authors would like to thank Manuel Bravo and Zhongmiao Li for their participation to refine the concept of Reversibility.

with reduced functionality. This can be acceptable for client-server applications, such as mobile applications that depend on a data center that remains a single point of failure. However, this is now changing as the Internet is becoming more and more decentralized: data centers are increasing in number and come in various sizes. Applications running on such an infrastructure need to have a decentralized architecture that is resilient to failure. Ideally, the application should survive with partial functionality during arbitrary system failures and recover its full functionality when the underlying system is restored. This is not just a fringe case: mobile and ad hoc networks, for example, have this kind of failure. Even supposedly stable parts of the Internet have peaks of unstable behavior.

We propose an approach to build applications able to survive arbitrary failures, providing reduced but predictable functionality in that case; and when the failures go away the application recovers its full functionality. We build on the concept of *Structured Overlay Network (SON)*, a known approach to building decentralized systems. We extend a SON to make it *Reversible*, which implies the system is able to regain its original functionality as the stress, e.g., churn or network partitioning, recedes. This paper focuses on one property of the network, namely *Churn*, i.e., nodes failing and being replaced by new correct nodes. We assume that churn varies over time and that the average number of correct nodes at any instant is constant. A SON that provides significant functionality at low churn, e.g., transactions over a key/value store, will no longer be able to do so at high churn. Applications that rely on transactions will no longer be able to use them. We want these applications to continue running nevertheless, with predictable behavior even with reduced functionality. Therefore the SON should inform the application of the provided functionality changes. Ideally, this should be done in a manner that works even for high churn. The SON can therefore not be relied on to do additional computation to determine its level of functionality. Under this constraint, is it possible for the SON to give useful information?

In order to describe the behavior of a SON, we introduce the concept of *Phase* of the system. Phases are well understood in physical systems [25]. We make an analogy for computing systems. A *Phase* is a subset of a system for which the qualitative properties are essentially the same. We consider a system as an aggregate entity composed of a large number of interacting parts, where parts are peers in our case. The phase is not a global property, but is observed separately at each node, and can be different for different nodes. No global synchronization and no extra computation is required to compute the phase; it is a direct consequence of the observed SON structure at each node. The phase of each node has a direct relationship with the available functionalities of the system. In contrast to stress, which is a global condition that cannot easily be measured by individual nodes, the phase is a local property that is directly known at each node. Thus, based on the current phase of a node, the application running on that node can manage its behavior in a stressful environment. As with the constituents of a physical matter, when external conditions change, each node of a SON changes phase independently. If that happens to many nodes,

we have a *Phase Transition* at system level. A *Critical Point* occurs when more than one phase exists simultaneously in significant fractions of a system.

Contributions: We define *Reversibility* and design our SON using the principles necessary to make it reversible. To our knowledge, no previous SON provides reversibility for the high values of churn we investigate. We demonstrate reversibility through simulation using realistic network conditions and churn varying over a large range. We present formal definitions of *Phase*, *Phase Transitions* and *Critical Points* in our context. We describe semantics of all the identified phases and sub-phases in our representative system. As a result of having a reversible system, we experimentally demonstrate reversible phase transitions: the nodes of the system change phase as the churn is varied. We present an API so that the application can access the current phase of a node and be notified when a phase transition occurs. Finally, we analyze the applications of these concepts towards the design of *Reversible* and *Predictable* systems. The overall contributions are as follows:

- Definition of *Reversibility*; First demonstration of a reversible SON and conditions to achieve the reversibility, under a wide range of churn values;
- Introduction of the concepts of *Phase*, *Phase Transition* and *Critical Point* in the context of a peer-to-peer network;
- Identification of different phases in a SON; Description of the semantics of all observable phases and sub-phases in the context of that SON;
- First demonstration of reversible phase transitions in a SON;
- An API to expose the current phase of a node, which gives local information about the stress, to the application;
- Analysis of how the application can use the phase concept to manage its behavior in a stressful environment.

The remainder of the paper is as follows. In Section 2 we describe a representative class of overlays. Section 3 presents the maintenance strategies of overlays. Section 4 defines and assesses reversibility of the maintenance strategies against churn. In Section 5, we study phase transitions in a SON, present an API to expose the current phase of a node to the application, and discuss the use of this knowledge. Section 6 discusses related work, and we conclude in Section 7.

2 Representative Overlays

We have chosen ring-based overlays, such as Chord [24], DKS [4], Beernet [20], as our representative systems, because the ring is competitive with other SON structures in terms of routing efficiency and failure resiliency [12]. In this section, we briefly discuss a model of ring overlays as per the reference architecture of [1].

A ring overlay has a virtual identifier space, I, which is a subset of $\mathbb{N}$ of size N. Each peer is associated with a unique id, $p \in I$, mostly using a uniform hash function or some random function. A peer with virtual identifier p is responsible for the interval $(predecessor(p), p]$, i.e., p is responsible for storing data items with keys $k \in (predecessor(p), p]$. Each peer p perceives I to be partitioned

into $\log(N)$ partitions, where each partition is k times bigger than the previous one. The routing table of p contains $\log_k(N)$ connections/fingers to some nodes from each partition. The neighborhood of a peer p, $N(p)$, is the set of peers with which p maintains a connection. For a target identifier i, peer p selects the closest preceding link, $d \in N(p)$ to forward the message. Since there are always k intervals, routing converges in $O(\log_k(N))$ hops.

Chord and Beernet: Chord [24] is the canonical ring-based SON. However, studies, e.g. [11], show that churn in Chord can introduce inconsistency. The reason is that the join/leave handling in Chord requires coordination of three peers that is not guaranteed due to non-transitive connectivity (i.e., A can talk to B and B can talk to $C \not\Rightarrow A$ can talk to C) on the Internet. In contrast to Chord, Beernet [20] does not assume transitive connectivity. This makes Beernet more resilient on Internet-like scenarios. Each step of join/leave handling in Beernet requires the agreement of only two peers, which is guaranteed with a point-to-point communication. Beernet has a correct lock-free three-step join operation, each step involving two peers. Lookup consistency is guaranteed after every step. As a result of such multi-step relaxed join operation, branches are formed: when a peer is not yet connected to its predecessor it forms a branch from the core ring. Fig 1 shows a Beernet network, where red (dark in B/W) nodes are organized into a ring and green (light in B/W) nodes are on branches. Due to branches, the guarantees about proximity offered by Beernet routing correspond to $O(\log_k(n)+b)$, where b is the distance to the farthest peer on the branch. We have used Beernet for experiments in this work.

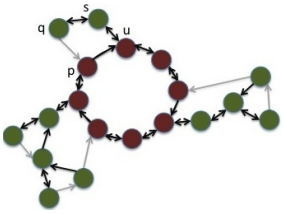


Fig. 1. Branches on a relaxed ring. Peers p and s consider u as successor, but u only considers s as predecessor. Peer q has not established a connection with its predecessor p yet.

3 Overlay Maintenance Strategies

As we consider "Reversible Phase Transitions" due to *Churn*, self-healing is crucial in order to be *Reversible*. This can be achieved by the maintenance strategy of an overlay. A *Maintenance Strategy* maintains the structural integrity of a SON while peers go offline or network connections fail. Several strategies are proposed in existing literature to achieve self-healing; Chord uses *Periodic Stabilization*, whereas DKS and Beernet rely on *Correction-on-Change*. Then there are gossip-based strategies, e.g., T-Man [13], which can construct and maintain a SON. So, why not just use gossip? Correction-on-change is much more efficient than gossip; whereas gossip is much more resilient. Therefore, we organize the maintenance strategies of overlays using *Efficiency* $\leftrightarrow$ *Resiliency* spectrum, as shown in Fig 2, where maintenance strategies at the top are efficient, but not resilient as churn increases, whereas the strategies at the bottom are resilient, however lack efficiency. Our goal is to get both efficiency and resiliency, so that a SON can achieve reversibility against extremely high churn. The philosophy behind our work is similar to that used by Plumtree [18].

Maintenance Strategy	Local/ Global	Reactive/ Proactive	Fast/ Slow	Safety	Bandwidth Consumption
Correction-on-*	Local	Reactive	Fast	Yes	Small
Periodic Stabilization	Local	Proactive	Slow	Lookup inconsistencies and uncorrected false suspicions can be introduced	High
Merger with Passive List	Global	Reactive	Adaptable	Yes	Adaptable
Merger with Knowledge Base	Global	Proactive	Adaptable	Yes	Adaptable

Fig. 2. *Efficiency* ↔ *Resiliency* Spectrum of Overlay Maintenance Strategies with their properties

As already mentioned, correction-on-change and correction-on-use (together referred to as *Correction-on-**), are efficient in terms of bandwidth consumption and rapid response against an event, however they fall short when the stress of the operating environment increases beyond a threshold (due to lack of liveness; without any event no maintenance is done). Correction-on-change handles join/leave/failure of nodes. Whenever a peer detects such events, it updates its neighborhood. Correction-on-use mainly corrects the fingers. Every time messages are routed, information is piggybacked to correct fingers. Thus, correction-on-use provides self-optimization and self-configuration, whereas partial self-healing is achieved through correction-on-change.

Gossip-based strategies are highly resilient against inhospitable environments, but costly in terms of bandwidth consumption and also, react slowly against an event. Using such strategies, each peer maintains a state (local knowledge of the overall system) and uses this knowledge to conduct maintenance. In our work, we have used a simple form of such strategies, *Knowledge Base (KB)* [21], where each peer maintains a best-effort view of the global membership of the system through listening only. The KB at each node can be accessed through an API.

Apart from these extremes, we organize the remaining strategies as per the spectrum: *Periodic Stabilization (PS)* can be seen as a weak form of gossip, where each node exchanges periodic messages with its successor to maintain its immediate vicinity. Such local corrections are able to achieve self-healing; however, might become a slow response while facing an inhospitable environment. As discussed in [10, 11], lookup inconsistencies and uncorrected false suspicions can be introduced in real implementations. Also, as per [15], for a low ratio of stabilization frequency to churn, while doing a lookup the longest finger of any peer is always found to be dead, which degrades routing efficiency. To avoid this, it is required to trigger PS often, making an inefficient use of bandwidth. Thus, presenting a trade-off, which is tuned by the period used for this strategy.

A similar bandwidth consumption-convergence time trade-off is added by *ReCircle* [23], in the form of a partition-merger. ReCircle has two parts: a PS algorithm and a *Merger*. The merger is triggered using a *Passive List (PL)*, where

each node maintains a list of suspected nodes and whenever a false-suspicion is detected, merger is triggered, thus restricting the gossip messages. The PL approach to trigger the merger is reactive to the operating environment. We have extended this in a proactive manner [21]: instead of PL, the merger is triggered periodically using KB; thus a resilient gossip-based maintenance.

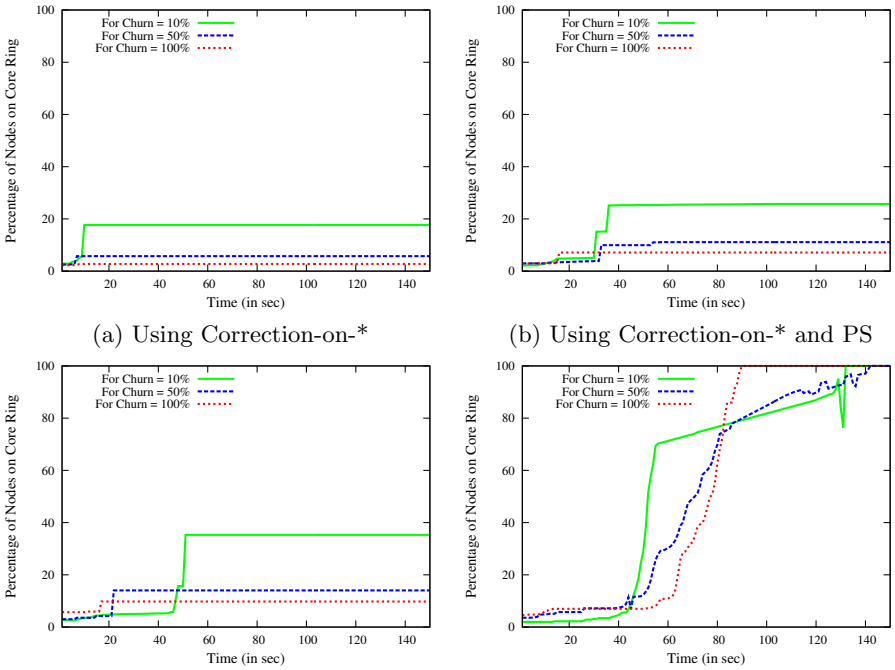
Apart from this spectrum, these strategies can also be classified along two dimensions: local/global and reactive/proactive. Following [2], we classify PS as a proactive and correction-on-* as a reactive mechanism. Using KB, as introduced in [21], the set of strategies covers all points in this two-dimensional space.

4 Reversibility and its Evaluation

Reversibility. A *Reversible* system is able to regain its original functionality as the external stress recedes. Given a function, $S(t)$, which returns the system stress as a function of time, in some arbitrary but well-defined units. A system is *Reversible* if there exists a function $F_{func}(id, S(t))$ such that the set of available operations of the system, $Op_{set} = F_{func}(id, S(t))$, and when $S(t) = 0$, the system provides full functionality at all nodes. Here, id is a node identifier and an operation is available for a given stress if the operation will eventually succeed. Reversibility is a related, but different property than *Self-Stabilization* [7]. A self-stabilizing system can repair itself from any arbitrary state. Reversibility does not assume anything about the system state. A self-stabilizing system is reversible, but a reversible system is not necessarily self-stabilizing.

Achieving Reversibility requires Knowledge Base. Reversibility is a nontrivial property. To our knowledge, no existing work demonstrates reversibility for a SON under continuous high churn. Our experimental results, presented in Fig 3, verify that the knowledge base is essential to ensure reversibility under continuous high churn. We assess reversibility in stepwise fashion, by integrating a new maintenance principle at each step and evaluating the behavior of the resulting system. We achieve reversibility only in the final step, shown in Fig 3d, which adds the knowledge base. We now explain these experiments in detail.

For our experiments, we have used a SON of 1024 peers. The underlying network is simulated by following the empirical distribution of minimum RTT provided in [3]. We have defined churn as percentage (%) of nodes turnover (nodes failing and being replaced by new correct nodes) per time unit (second in this work). During the steady state of the SON we inject 10%, 50% and 100% churn for 1 minute. The churn events are modeled as a *Homogeneous Poisson Process (HPP)* with λ events/sec, where $\lambda = \frac{2*C*1024}{100}$ for C% churn. After withdrawing churn, we observe the SON's self-healing with time. To quantify self-healing, we have used the metric: % of nodes on the core ring. We find out the maximal ring in the system and report % of nodes on it. The ultimate goal is to have the metric converge to 100%. A fixed workload is used by injecting transactions, modeled as a HPP with $\lambda = 1$ transaction/sec. A transaction reads one key and updates another one. Starting from the withdrawal of churn, for each second we present % of nodes on core ring and an average of 20 independent



(a) Using Correction-on-*

(b) Using Correction-on-* and PS

(c) Using Correction-on-*, PS and Merger with Passive List

(d) Using Correction-on-*, PS and Merger with KB

Fig. 3. % of nodes on the core ring as a function of time (in sec) after withdrawing churn to assess reversibility. Fig 3a, 3b and 3c are not reversible (nodes on the core ring never converges to 100%). Fig 3d using KB is reversible.

runs are taken for each second. We remark that the apparent termination time of an experiment in Fig 3d is the maximum among the samples used.

As Fig 3a shows correction-on-* fails to achieve reversibility even for the lowest intensity (10%) of churn used in our experiments. After withdrawing churn, the structure of the system remains almost the same. This is due to the lack of liveness of these principles, thus exhibits very limited reversibility.

In Fig 3b we can see improvements after integration of PS; however the system is still unable to achieve reversibility. The period used is 3 seconds.

As Fig 3c shows, the integration of merger with PL does not show much improvement over the combined local healing. The reason is the existence of isolated nodes in the system (explained below). The nodes on the overlay have no reference to these nodes. So, adding reactive merge does not achieve reversibility.

Why not reversible yet? As we can see in Fig 3c the system is still not reversible. As our investigation shows, there are peers whose joining fails under churn. The first step of joining is to do a *lookup* for successor and after receiving a response a new peer becomes part of the SON. For a join to fail either the lookup request is lost while routing or the successor has failed after receiving the lookup request. If we ignore the processing time at successor, then $P(join\_failure) \propto P(lookup\_failure)$. Fig 4 shows % of incomplete lookups and joins for varying

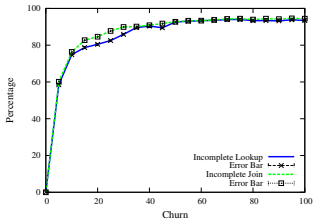


Fig. 4. % of incomplete lookups and joins after 1 min. churn injection

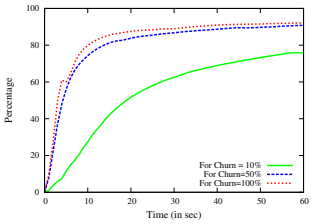


Fig. 5. % of incomplete joins with time during injection of churn for 1 min.

churn. We have used the same experimental setup with lookup requests as a HPP with $\lambda = 100$ requests/sec. We also present the accumulation of pending join requests with time in Fig 5, especially for high churn. As is evident, high churn makes the overlay unstable, which does not allow new peers to join.

As shown in Fig 3d, after the integration of KB the system achieves reversibility. In order to ensure successful joining of new nodes, we have extended nodes with repeated join attempts (until a response is received) with new join references, which are provided by KB. In our experiments, if a node is unable to join with its current join reference within 90 sec (a tunable parameter that will be referred to as *Join Timeout*), it requests a new join reference from the application layer. The application layer provides a new join reference by accessing and accumulating the distributed KB, or using a previously cached one. The isolated peer then triggers a new join request with that. We have chosen a conservative value of 90 sec for *Join Timeout* to avoid triggering of unnecessary repeated join requests. This parameter can be adapted based on the operating environment and RTT distribution of the underlying network, which is left as future work. Along with this, we have used the proactive merger using KB. In some runs we have observed partition of the system after the isolated nodes complete their join procedures. For these scenarios, the PL approach used in [23] fails to trigger the merging. In order to merge such partitions proactive merger using KB is required. As we can see in Fig 3d, the system achieves reversibility.

To summarize the outcome of our experiments: *Efficient* maintenance strategies fail to achieve reversibility as churn increases; a *Resilient* maintenance strategy is required to make the system reversible in case of extremely high churn.

5 Phase, Phase Transitions, API and Application

In this section we formally define *Phase*, *Phase Transition* and *Critical Point* in our context. We describe the semantics of all identified phases and sub-phases in our representative system. We relate our phase concept to Reversibility. We empirically demonstrate reversible phase transitions in a reversible system. We present an API to give useful phase information to the application layer and discuss various applications of this knowledge.

5.1 Definition of Phase, Phase Transition and Critical Point

We present formal definitions of our concepts by drawing an analogy with these terms in physical systems [25]. This analogy is introduced to make it easier to understand intuitively what phase means. We consider a system, $S = \{S_1, \ldots, S_n\}$,

where each $S_i : 1 \leq i \leq n$, is an interacting part of S. In our case S is a SON and S_i is a node of the SON. The system is partitioned into k subsets, such that: (1) $P_1 \uplus P_2 \uplus \ldots \uplus P_k = S$, (2) For any $P_i : 1 \leq i \leq k$, qualitative properties are the same $\forall S_x \in P_i$, (3) For $P_i, P_j : i \neq j$, qualitative properties are different, i.e., if F is a function of qualitative property, then $\forall_{i,j:i\neq j} \implies \forall S_x \in P_i, \forall S_y \in P_j : F(S_x) \neq F(S_y)$. We can say each $P_i : 1 \leq i \leq k$ is an observed phase in S. A *Phase Transition* at system level occurs when a significant fraction of a system's parts change phase. This can happen if the local environment changes at many parts. A *Critical Point* occurs when: (1) $k > 1$, (2) at least two, i_1 and i_2, such that $\mid P_i \mid \gg 1$. This paper investigates phases and phase transitions in a SON with varying churn.

Reversibility and Phase: We introduce the concept of phase so that applications can manage their behavior in stressful environments. We have defined the *phase* P_i at each node i to be a well-defined local property of the node. We define the *phase configuration* of the system to be the vector $P_c = (P_1, P_2, P_3, ..., P_n)$. Both phases and phase configurations are functions of time. We can define Op_{set} (See Section 4) in terms of them: $Op_{set} = F_{det}(id, P_c(t))$.

In the case of Beernet, we can determine a property that satisfies the formal definition of phase given above. The phase of a node is clearly determinable at that node: there are three mutually exclusive situations depending on neighbor behavior (neighbors on core ring, neighbors on branch, no neighbors). There is an analogy between these three phases and the solid, liquid, and gaseous phases in physical matter (e.g., water). Also, when a node is on a branch (i.e., liquid phase), we can identify three sub-phases in terms of available functionalities and probability of facing an immediate phase transition. We define semantics of each phase and sub-phase, in analogy with the solid, liquid, and gaseous phases in physical matter and translate them in terms of available functionalities.

- **Solid (P_S):** The solid state of a matter is characterized by structural rigidity, where atoms or molecules are bound to each other in a fixed structure. In case of SONs, if a peer has stable predecessor and successor pointers (i.e., the peer is on core ring), along with a stable finger table, then it can be termed to be in solid phase. It can be safely assumed that such a peer can support efficient routing, thus accommodate up-to-date replica sets, thus leading to all the upper layer functionalities, e.g., transactional DHT.
- **Liquid:** A thermodynamic system is in the liquid state where molecules are bound tightly but not rigidly (neighbors can change). In case of SON, if the peer is on a branch, it is less strongly connected than those in P_S; thus in liquid phase. However a peer can be on a branch temporarily, e.g., as part of the join protocol or due to a false suspicion as a result of sudden slow-down of the underlying physical link. We identify three liquid sub-phases.
  - P_{L1}: If the peer is on a branch, but the depth of the peer (distance from the core ring) is ≤ 2. Also, the peer still holds a stable finger table. The justification of depth of 2 for this sub-phase is based on the evaluation of average branch sizes in [20], where it is shown that the average branch size of Beernet is ≤ 2, corresponding to the connectivity among peers

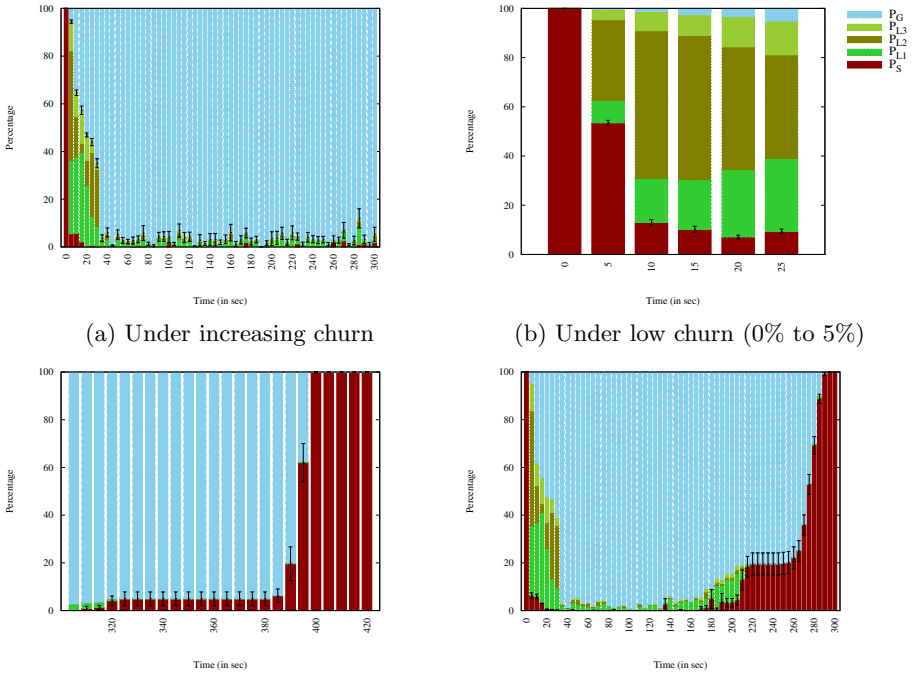
on the Internet. So, if a peer's depth on a branch is ≤ 2, the operating environment from a peer's perspective is still the usual one, it might temporarily be pushed on a branch. From an application's perspective, the peer is still able to provide all the higher layer functionalities.
  - P_{L2}: If the peer is on a branch with a depth > 2, but it is not the tail of the branch. Also, the finger table at the peer still holds $> 50\%$ valid fingers. So, the peer is still able to support at least all DHT operations, however successful transactions are not guaranteed anymore.
  - P_{L3}: If the peer is on a branch with a depth > 2 and it is the tail of a branch (farthest node from the core ring). The tail of a branch has higher probability to get isolated during churn, thus introducing unavailability in the key range [20]. Also, most of the fingers in the peer's finger table are invalid or crashed. From a application's perspective, the peer in this sub-phase provides very limited functionality, mostly basic connectivity.
- **Gaseous (P_G):** The gaseous state of matter is made up of individual molecules that are separated from each other. When Beernet experiences high churn, at some point the system is completely dissolved, resulting in isolation, thus gaseous phase, of all nodes. In this work, we have considered only extreme case of partitioning of the system. However, in practice, if there are > 1 nodes per physical machine and the network breaks down, then small ringlets will be formed by the nodes on the same physical machine. For such scenario, the nodes are not completely isolated, however in another form of gaseous phase. The investigation about such gaseous sub-phases is left as future work.

5.2 Observation of Phase Transitions

We show experimentally the existence of phase transitions in our representative reversible system as the churn intensity varies. For this we have used similar experimental setup as described in Section 4 with a network of 1024 peers. We have measured the percentages (%) of nodes in different phases and sub-phases.

Increasing Churn with Time We study phase transitions under increasing churn and reverse transitions when churn is removed. We start with 5% churn and increase the intensity by 5% every 5-second for 5 minutes. After that churn is withdrawn; we let the system run until the completion of self-healing (i.e., perfect ring). Every 5-second we take a snapshot of the system, i.e., the percentages of nodes at each phase and sub-phase throughout a run. We have used mean value for 20 independent runs. Fig 6a and Fig 6c show the states of the system during increasing churn followed by zero churn respectively.

In Fig 6a (error bars for P_G only) the red area of each bar corresponds to % of nodes which are in phase P_S. At time 0, i.e., starting of the experiment, all nodes are organized into a perfect ring. As churn is increased nodes start moving on branches, the green area of each bar, these are the nodes which are in liquid phase. We have also identified nodes on branches which have different liquid sub-phases, as per the semantics described before. We can figure out some



(a) Under increasing churn

(b) Under low churn (0% to 5%)

(c) After withdrawing churn

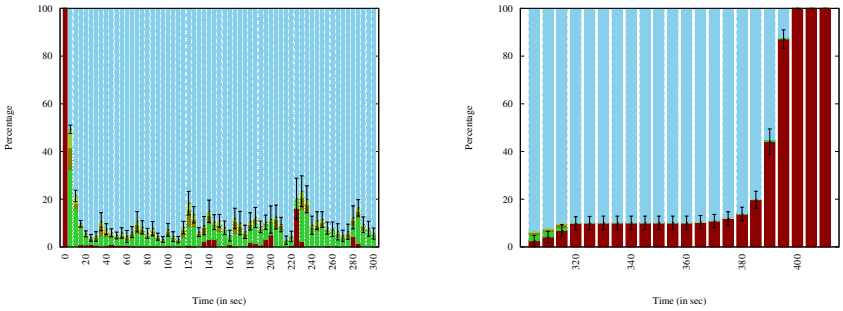
(d) Under increasing and decreasing churn

Fig. 6. Phase Transitions in Beernet: red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to % of nodes in solid, liquid and gaseous phases respectively

trends apparent in Fig 6a. For example, 30% of churn is a critical value, observed at 30 sec, as a significant fraction of nodes change from liquid to gaseous phase.

The solid to liquid transition happens between 0% and 5% churn. In Fig 6a, we can see a sharp fall of % of nodes on core ring from 0 to 5 sec. In order to analyze this transition we have zoomed into this area. For this experiment, during steady state of SON, we start with 1% of churn and every 5-second we increase churn intensity by 1% till we reach 5% of churn. Also a snapshot is taken every 5 second as before. Fig 6b shows the result. We have used mean values of 20 samples and only error bars for the P_S are shown. As we can see, as the churn intensity is increased from 1% to 2%, during 5 to 10 seconds, a large fraction of nodes changes phase from solid to liquid.

Fig 6c (error bars for P_S only) shows the recovery of the SON after churn is withdrawn (i.e., $6-9$ minutes of our experiment). Here starting with all isolated nodes, a small fraction of nodes changes to transient liquid phase. We can see a period of about 90 sec, during which the % of isolated nodes remains same. The reason is the *Join Timeout* parameter (see Section 4), which is set as 90 sec. The transition from gaseous state is controlled by this tunable parameter. Finally all nodes are self-organized into a perfect ring, solid state of SON, within 400 sec.



(a) Under continuous churn of 30% (b) After withdrawal of 30% churn

Fig. 7. Phase Transitions in Beernet: red, green (different shades corresponds to 3 liquid sub-phases) and blue (dark, gray and light-gray in B/W) areas correspond to % of nodes in solid, liquid and gaseous phases respectively

Continuous Moderate Churn We seek answer to the question: whether a phase transition happens in a SON at continuous moderate churn. For this experiment we have chosen churn equal to 30%, as we have observed that 30% of churn is a critical value. During steady state of SON, we start injecting 30% churn for 5 minutes. Then churn is withdrawn and we let the SON do self-healing until all nodes are on the core ring. During our experiment, we take measurements every 5-second and present mean value of 20 independent runs in Fig 7a (error bars for only P_G) and Fig 7b (error bars for only P_S). As we can see in Fig 7a, a significant fraction of nodes change phase during first 10 seconds, thus justifying our deduction that churn intensity of 30% to be a critical point. Also, we notice that, as in a thermodynamic system (e.g., water), it takes longer time for the system to reach a gaseous state, in fact there is no clear transition where all nodes are in gaseous phase. During injection of 30% churn for 5 minutes, a small fraction of nodes remains in solid or liquid phase surrounded by the remaining large fraction be in gaseous phase. The reverse transition shown in Fig 7b follows the same pattern as in Fig 6c.

Gradual Increase and Decrease of Churn Till now, we have withdrawn churn completely; what behavior does the system exhibit if the intensity of churn is gradually decreased? During steady state of SON, we start injecting 5% of churn and increase the intensity of churn every 5-second until churn is of 100%. Then we gradually decrease churn by 5% every 5 second until it reaches 0. We take measurements every 5 second throughout the experiment and present mean values of 20 independent runs in Fig 6d (error bars for only P_S). The behavior follows our previous deductions. Around 30% of churn a significant fraction of nodes change phase. Between 40 and $100 - 105$ seconds we see a gaseous system. During gradual decrease of churn intensity, there is increasing connectivity among nodes, followed by organization into ring structure, evidential of reversible phase transitions in our system due to increasing and decreasing churn.

5.3 API for Phases and Phase Transitions

Since phase is a node-specific property, an API exists on each node to expose its phase to the application layer. Next, we describe the use of this knowledge in real application scenarios. In our future work, we intend to empirically demonstrate them. Our API supports push and pull methods to communicate the current phase of a node. A node can be in one of the phases described in Section 5.1.

- ***getPhase(?P_{cur})*** Binds P_{cur} to the current phase of the peer.
- ***setPhaseNotify(f)*** Sets a user-defined function, ***f(?P_{new})*** to be executed when the phase changes. P_{new} is bound to the next phase of the peer and ***f*** is executed. Executions of ***f*** are serialized in the same thread over a stream of successive phases.

The application running on top of a SON can use the phase information to make the system reversible and predictable. We illustrate the usefulness of the phase concept using a real application scenario. Consider a *Distributed Version Control System* running on top of SON, which is notified that the underlying node changes its phase. The application notifies the user via an indicator, B_{conn}, that changes its color to indicate the phase of the node. B_{conn} can be green or yellow or red, denoting respectively *solid/liquid/gaseous* phase of the node. Suppose the user uses this system on a network having intermittent connectivity (e.g., Wi-Fi on a fast train). As long as B_{conn} is green, the user can continue her work without being concerned. However, as B_{conn} changes to yellow, the user can initiate a *pull* to retrieve the most recent version and *push* her own changes. These allow the user to work productively offline on the up-to-date version and prevent any potential data-loss. Thus, the application itself can achieve reversibility as the connectivity of the underlying network is restored, and the user is able to predict the behavior of the system. For example, the application (e.g., real-time collaborative editing) can adapt the philosophy of exponential back-off as TCP congestion algorithm, which is now brought up to the user level in terms of phase. Further, the underlying node can adapt its maintenance to the current phase. For example, when a node is in P_S phase, an efficient strategy like correction-on-* is sufficient. As the node faces a transition to P_{L1}, it can turn on a more resilient strategy, like PS.

6 Related Work

We briefly summarize the relevant work on self management in SONs and phase transitions. Krishnamurthy et al. in [16] use fluid model approach to analyze the probability of network disconnection and the fraction of incorrect pointers (successor and fingers) in Chord under churn. In their follow-up work [17], they use master-equation of physics to do comparative analysis of periodic stabilization and correction-on-change under churn. Another analytical work [19] establishes a lower bound on the maintenance rate of a SON under churn in order to remain connected. In [8] and [9], a physics-inspired approach is used

to analyze performance of Chord and also investigate about intensive variables (i.e., variables independent of system size) related to self-organization and self-repair. Design decisions such as self-tuning mechanisms are described in [5] for self-organization/self-adaptation of overlay networks. The analytical framework in [14] can be used to characterize the routing performance of SON under churn. Our empirical study can be seen as complementary to these analytical works.

Diligent search has failed to uncover any empirical work on phase transitions in SONs. However, we have found one analytical work [15] carried out for Chord that shows a critical point in the parameter space at which the system with high probability breaks down, i.e., efficient routing becomes impossible. Such phase transitions happen due to high churn and large link delays, resulting in a finite fraction of the connections to be always incorrect. In [6] phase transitions in unstructured P2P network are studied to identify resource-efficient operating points for various global properties. For power-law networks, [22] presents a decentralized monitoring algorithm where each node estimates global statistical parameters and influences them to optimize relevant network characteristics.

7 Conclusion

As *Structured Overlay Networks (SONs)* are a popular choice to implement large-scale distributed software systems, it is important to ensure their reversibility against harsh environments. We have defined *Reversibility* and experimentally demonstrated a reversible system. We have identified the necessary maintenance principles to achieve reversibility against *Churn*. We have proposed the concepts and semantics of *Phase*, *Phase Transition* and *Critical Point* in our context. Also, we show that a reversible system does reversible phase transitions, i.e., it "boils" to the gaseous state (becomes disconnected) when churn increases and "condenses" from gaseous back to solid phase as churn intensity goes down. We also identify the apparent "critical points" from the experiments while doing such transitions. Finally, we have presented an API to make the phase of a node explicit to the application layer and analyze the applications of our concepts of phase and phase transitions toward designing predictable and reversible systems.

This paper is only the first step; we intend to investigate further the analogy between phase in SONs and in physical systems. We will design an application that take advantage of our API to survive in extremely hostile environments. We also intend to gain more insights about the maintenance strategies.

References

1. K. Aberer, L.O. Alima, A. Ghodsi, S. Girdzijauskas, M. Hauswirth, and S. Haridi. The essence of P2P: a reference architecture for overlay networks. In *Proc. P2P'05*.
2. K. Aberer, A. Datta, and M. Hauswirth. Route maintenance overheads in DHT overlays. In *Proc. WDAS*, 2004.
3. J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proc. ACM SIGCOMM IMC*, 2003.

4. L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *Proc. CCGrid*, 2003.
5. S. Apel and K. Böhm. Self-organization in overlay networks. In *CAiSE Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA)*, 2005.
6. F. Banaei-Kashani and C. Shahabi. Criticality-based analysis and design of unstructured peer-to-peer networks as "complex systems". In *Proc. CCGrid*, 2003.
7. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), November 1974.
8. S. El-Ansary, E. Aurell, P. Brand, and S. Haridi. Experience with a physics-style approach for the study of self properties in structured overlay networks. In *SELF-STAR: Intl. Workshop on Self-* Properties in Complex Info. Sys.*, 2004.
9. S. El-Ansary, E. Aurell, and S. Haridi. A physics-inspired performance evaluation of a structured peer-to-peer overlay network. In *Proc. PDCN*, 2005.
10. M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. WORLDS*, 2005.
11. A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, KTH, Sweden, 2006.
12. K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM*, 2003.
13. M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Proc. ESOA*, 2005.
14. J. S. Kong, J. S.A. Bridgewater, and V. P. Roychowdhury. Resilience of structured P2P systems under churn: The reachable component method. *Comput. Commun.*, 31, 2008.
15. S. Krishnamurthy and J. Ardelius. An analytical framework for the performance evaluation of proximity-aware structured overlays. Technical report, SICS, Sweden, 2008.
16. S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi. An analytical study of a structured overlay in the presence of dynamic membership. *IEEE/ACM TON*, 2008.
17. S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi. Comparing maintenance strategies for overlays. In *Proc. PDP*, 2008.
18. J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *SRDS'07*.
19. D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proc. PODC*, 2002.
20. B. Mejías. *Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage*. PhD thesis, UCL, Belgium, 2010.
21. R. R. Paul, P. Van Roy, and V. Vlassov. Interaction between network partitioning and churn in a self-healing structured overlay network. In *Proc. ICPADS*, 2015.
22. I. Scholtes, J. Botev, A. Höhfeld, H. Schloss, and M. Esch. Awareness-driven phase transitions in very large scale distributed systems. In *Proc. SASO*, 2008.
23. T. M. Shafaat. *Partition Tolerance and Data Consistency in Structured Overlay Networks*. PhD thesis, KTH, Sweden, 2013.
24. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.
25. Wikipedia. Phase (matter). `https://en.wikipedia.org/wiki/Phase_(matter)`, 2016.