Designing Distributed Applications Using a Phase-Aware, Reversible System

Ruma R. Paul*
Microsoft
Dhaka, Bangladesh
ruma.paul@microsoft.com

Jérémie Melchior, Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
{*jeremie.melchior, peter.vanroy*}*@uclouvain.be*

Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
vladv@kth.se

***Abstract*—Distributed applications will break down or perform poorly when there are too many failures (of nodes and/or communication) in the operating environment. Failures happen frequently on edge networks including mobile and ad hoc networks, but are also unexpectedly common on the general Internet. We propose an approach for designing stress-aware distributed applications that can take environment stress into account to improve their behavior. We give a concrete illustration of the approach by targeting applications built on top of a *Structured Overlay Network (SON)*. Our underlying SON is *Reversible* and *Phase-Aware*. A system is *Reversible* if the set of operations it provides is a function (called the *reversibility function*) of its current stress (i.e., all perturbing effects of the environment, including faults), and does not depend on past stress. Reversibility generalizes standard fault tolerance with nested fault models. When the fault rate goes outside the scope of one model, then it is still inside the next one. In order to approximate the reversibility function we introduce the concept of *Phase*, which is a per-node property that gives a qualitative measure of the available system operations under the current stress. Phase can be determined with no extra distributed operations.**

We show that making the phase available to applications allows them to improve their behavior in environments with high and variable stress. We propose a Phase API and we design an application, a collaborative graphic editor, that takes advantage of phase to enhance self-adaptation and self-optimization properties. Furthermore, we analyze how the application itself can achieve reversibility in the application-level semantics. Using the phase of the underlying node, the application provides an indication to the user regarding its behavior. Thus, the application has improved behavior with respect to the user, i.e., the user can better understand and decide what to do in a high-stress environment.

***Keywords*-reversibility; phase; high-stress environment; reversible system; phase-aware application**

I. Introduction

Too much stress (e.g., node and/or communication failures, slow communication) in the operating environment can cause distributed applications to break down or perform poorly, without providing any prior indication to the users. For such scenarios, the application may switch to an "offline mode" with reduced functionality. This may 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 with increasing decentralization of the Internet: data centers are increasing in number and come in various sizes. Also, the technological advances and always-increasing user demands of better QoS and scalability of distributed applications are preparing the way for a paradigm shift in the field. We are moving away from data-center-based computing toward edge computing, i.e., pushing the computation towards the logical extreme of a network: mobile edge computing, cooperative distributed peer-to-peer ad-hoc networking, Internet of Things (IoT)—to name a few of the wide range of technologies covered by edge computing. Ideally, the applications running on such increasingly decentralized infrastructures should: (1) survive with partial functionality during arbitrary stress and recover their full functionality when the stress recedes; and (2) change their behaviors to take functionality loss into account and make themselves predictable to the user.

Reversibility. We propose an approach to design applications capable of surviving arbitrary stress, providing reduced but predictable functionality in that case; and, when the stress goes away, fully recovering functionality. We apply our approach to applications designed using Structured Overlay Networks (SON), a well-known approach for building decentralized distributed systems. In previous work, we have defined *Reversibility* of a representative SON against churn (i.e., nodes failing and being replaced by new correct nodes) [1] and network partitioning [2]. A system is *Reversible* if the set of available operations (i.e., the functionality of the system) is a function of its current stress (called *reversibility function*) and does not depend on the history of the stress. In this paper, we expand and refine this idea and present *two different forms of Reversibility*. Also, this paper focuses on designing applications on top of reversible systems so that, taking advantage of the reversibility of the underlying system, the application can manage its behavior while experiencing stress in its environment. We analyze the design decisions of the application that will allow it to have enhanced self adaptation and self optimization, and achieve reversibility at the application level as well as the SON level.

*This work was done during the author's doctoral study at UCL, Louvain-la-Neuve, Belgium and KTH, Stockholm, Sweden

Phase API. A SON providing functionality at low stress, e.g., transactions over a *Distributed Hash Table (DHT)*, might no longer do so at higher stress. Applications relying 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. For this purpose, an application needs to be informed by the SON about the qualitative changes in the provided functionality. This information is contained in the reversibility function. In order to practically approximate this function, we introduce a *Phase* concept [1], analogous to that in physical systems [3]. In contrast to stress, which is a global property and cannot easily be measured by individual nodes, phase is a per-node property. All nodes with the same phase exhibit the same qualitative functionality, which is different for nodes in different phases. The phase is provided to the application at each node through a Phase API. This allows applications to manage their behaviors predictably in stressful environments.

Contributions. The contributions are as follows.

- Extension of the *Reversibility* concept; Introduction of *two* different forms of reversibility;
- Definition and computation of self-awareness at each node and how a self-aware node determines its phase;
- Approximation of available functionality (reversibility function) at each node based on its phase;
- First demonstration of a *Phase-Aware* application design and its evaluation;
- Demonstration of self-adaptive/optimization behavior of the application under churn, delays, and partitions;
- Demonstration of how functionalities of an application operating under stress can be predictable to the user.

Structure of the paper. Section II defines the concepts of Reversibility and Phase. Section III presents our representative system, a reversible SON. Section IV uses the phase concept to approximate the reversibility function in our SON, and explains how we extend a peer to determine its current phase. Section V shows a phase-aware application design based on an existing distributed transactional editor called *DeTransDraw* (DTD). Section VI evaluates and analyzes the enhancements in DTD due to the phase-oriented design. Section VII discusses related work, and we conclude in Section VIII.

II. Reversibility and Phase

In [1] we have presented a formal definition of *Reversibility*. Here, we refine that definition and discuss *two different forms of reversibility*. We also discuss how reversibility is related to fault tolerance and self stabilization. We define stress as a measure of all the potential perturbing effects of the environment on the system, including both faults and other nonfunctional properties such as communication delay. We conclude with a brief definition of the *Phase* concept and its relation with reversibility.

A. Reversibility

Given a function $S(t)$ that returns system stress as a function of time, in some arbitrary but well-defined units. For example, $S(t)$ can represent network partitioning as a function of time. $S(t)$ can return any data structure needed to represent the stress.

A system is *Reversible* if there exists a *total* function $F_{rev}(s)$, which we call the *reversibility function*, that returns the set of available system operations $Op_{avail} = F_{rev}(s)$ for stress s; and when there is no stress, i.e., $s = 0$ (where the symbol 0 indicates no stress), then the system provides full functionality at all nodes. From this definition, we can identify *two* different forms of reversibility.

- **Weak Reversibility.** A system is *Weakly Reversible* if for all t such that $S(t) = 0$, the system provides full functionality at all nodes. An operation is guaranteed to complete if $S(t) = 0$ during the operation.
- **Strong Reversibility.** A system is *Strongly Reversible* if there exists a function $F_{rev}(s)$ such that the set of available operations of the system is equal to $F_{rev}(s)$ for any value of stress s. An operation is *available* for a given stress if it eventually succeeds (i.e., it will fail only a finite number of times if tried repeatedly, and then succeed).

For strong reversibility, if the stress changes from s_1 to s_2, and then back to s_1, then the functionality that was available at s_1 is regained. For weak reversibility, s_1 is always 0, and nothing is said for nonzero stress. Section II-B presents our approach to approximate the reversibility function.

Reversibility is a related but different property than *Fault Tolerance*. A fault-tolerant system is resilient for a given fault model, but its behavior is undefined outside that model. In order to specify a system that can provide partial functionalities under stress, we would have to define a separate fault model for each of those partial functionalities. By introducing reversibility we generalize fault tolerance with nested fault models. When the fault rate goes out of the scope of one model, it is still in the next model.

Self Stabilization was introduced in [4] as a non-masking fault tolerance for distributed systems. Unlike self stabilization, reversibility does not assume anything about system state. A self-stabilizing system survives any temporary perturbation of its state, and it will always eventually return to a valid state when there are no perturbations. In contrast, a reversible system provides predictable functionality under long or continuous stress; therefore, the reversibility property is more useful in practice. A self-stabilizing system is reversible, but a reversible system is not necessarily self-stabilizing. However, self stabilization implies weak reversibility. Weak reversibility may imply self stabilization in some cases, if the values of $S(t)$ can cause the system's state to be any arbitrary value.

B. Phase

In order to approximate the reversibility function we introduce the concept of *Phase*. In contrast to stress, which is a global condition that cannot easily be measured by individual nodes, the phase is a local property of a node. A *Phase* is a subset of a system for which the qualitative properties are essentially the same. The phase P_i at the node i is a well-defined local property of the node. As a node experiences changes in its local environment, it changes its phase independently of the other nodes. If this happens at many nodes, we have a *Phase Transition* at system level.

We define the *Phase Configuration* of the system as the vector $P_c = (P_1, P_2, \ldots, P_n)$. Both phase and phase configuration are functions of time. If P_c contains enough information, then we can define Op_{avail} (Section II-A) in terms of it: $Op_{avail} = F_{rev}(S(t)) \approx F_{phase}(P_c(t))$. The *phase function* F_{phase} is total: $\forall P_c(t), F_{phase}(P_c(t)) \mapsto Op_{set}$, here Op_{set} can be a vector, where an element is a set of available operations at a particular node. In Section IV we apply the the Phase concept to our representative system to estimate the phase function.

III. A Reversible System

We have used a ring-based SON, namely Beernet [5], which hosts a transactional key-value store, as the underlying system in our work. Beernet is based on Chord [6]. Unlike Chord, Beernet has correct lock-free join/leave operations, and we have extended Beernet to be reversible. We have chosen ring-based overlays, because the ring is competitive with other SON structures in terms of routing efficiency and failure resiliency [7]. We briefly describe a model of ring overlays following the reference architecture of [8].

Ring overlay. A ring overlay has an identifier space, $I \subseteq \mathbb{N}$, of size N. Each node is associated with a unique id, $p \in I$, using a hash function or some random function. A node with an id p is responsible for $(predecessor(p), p]$, i.e., p is responsible for keys in the range $k \in (predecessor(p), p]$. Each node 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 (referred as *fingers*) to some nodes from each partition. The neighborhood of 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.

Beernet, A reversible Chord. Beernet is a version of Chord, the canonical ring-based SON, with correct lock-free join/leave handling protocols. 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 does not assume transitive connectivity. This makes Beernet more resilient on Internet-like scenarios. We briefly describe Beernet and how we have extended it to achieve reversibility.

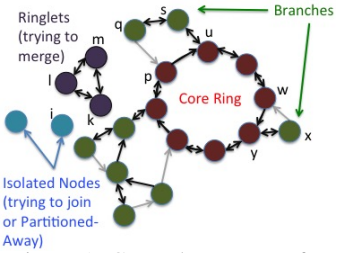


Figure 1: General structure of a relaxed ring

To avoid locking, the join/leave operation in Beernet is done in two steps. Each step is a one message round trip between two nodes. This greatly simplifies the join/leave since it does not have to handle special cases for failure. A consequence of such multi-step operations is that the network is structured as a core ring surrounded by branches. The branches contain nodes that have done one step of the procedure. Fig 1 shows a general structure of Beernet, where red (dark in B/W) nodes are organized into a ring, green (gray in B/W) nodes are on branches and blue (light-gray in B/W) nodes are isolated. Ringlets are formed (by k, l, and m in Fig 1) due to logical partition of the system due to stress, e.g., high churn or physical partition. In Beernet, unlike Chord, a node always forwards a message to the responsible node in order to take into account any branch in between, and if a node considers its successor to be responsible, it sets a flag in the message. 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.

For Beernet, which hosts a transactional key-value store, we can identify an ordered set of functionalities, $Op_{total} = \{TR, Op_{DHT}, Routing_{efficient}, Routing_{basic}, NONE\}$.

- TR = Transactional functionality.
- Op_{DHT} = Basic DHT operations, $get(Key, ?Value)$ (binds $Value$ to the value stored with key Key) and $put(Key, Value)$ (stores the value $Value$ associated with key Key). These operations access the storage of the noderesponsible for the corresponding key;
- $Routing_{efficient}$ = Logarithmic routing;
- $Routing_{basic}$ = Basic routing by following the successor pointer of each node, i.e., $O(n)$ routing.
- $NONE$ = No functionality.

In our case, Op_{total} is ordered, i.e., if the system supports transactional functionality (which is the highest-level) then all the lower-level functionalities are also available. Following the definition of reversibility (see Section II-A), the set of available operation at a node p, $Op_{avail}[p] \subseteq Op_{total}$.

Maintenance strategies to achieve reversibility. We have extended Beernet to be reversible. In order to be reversible complete self healing is crucial. 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. We have organized the maintenance strategies of overlays using the *Efficiency* $\leftrightarrow$ *Resiliency* spectrum [1]. Correction-on-

change/use, as used in DKS [9], is much more efficient than gossip-based strategies (e.g., T-Man [10]); whereas gossip is much more resilient. We have shown that both efficiency and resiliency are required for reversibility. A full discussion of the maintenance strategies used in Beernet for reversibility is outside the scope of this paper; we refer interested readers to [1], [2]. Reversibility is useful in other distributed systems as well, e.g., in the Lasp implementation [11], which uses convergent data structures and is naturally reversible.

IV. Approximating the Reversibility Function

The goal of this section is to approximate the reversibility function, i.e., to compute the phase function for our representative system. In this paper, we do this based on an assumption: the rest of the system is qualitatively the same as the current node, i.e., at node i, $P_c[j : j \neq i](t) = P_i(t)$. Thus, each node i computes the phase function, thus approximates Op_{avail} based on its current phase. The reason behind such simplification is to avoid any extra distributed operation or bandwidth consumption. In order to have a better approximation, each node needs to maintain a *Phase Base (PB)* (a local view of P_c) and communicate with others to enhance its PB. This can be done by piggybacking PB with messages routed via each node to avoid any extra distributed operation; however still consumes extra bandwidth. The implementation of PB, and the investigation about extra resource consumption to justify the worthiness of such protocol, are left as future work.

In this section, we explain how we apply the definition of phase (see Section II-B) to identify different phases, sub-phases in Beernet. Next, we define the semantics of each phase and sub-phase in terms of available operations, i.e., we compute the phase function based on our simplifying assumption described above and expose Op_{avail} to the application layer by exposing the phase of each node. We also explain how we have made each peer self-aware so that a peer can determine its current phase with high confidence without any global synchronization.

A. Phases in Relaxed Ring

In case of Beernet, we have identified a property that satisfies the definition of phase. Depending on the neighbors' behaviors of a node there are three mutually exclusive states: neighbors on core ring, neighbors on branch, and no neighbors. This, together with the validity of the fingers in the routing table of a node, gives detailed information about the structure of the ring and the expected routing guarantees. Drawing an analogy with the phases in physical matter (e.g., water), we have termed these three phases as *solid*, *liquid* and *gaseous* respectively. Also, when a node is on a branch (i.e., liquid phase), we have identified three sub-phases in terms of available operations and probability of facing an immediate phase transition. All these phases and sub-phases are distinguishable from each other and provide sufficient information to understand the available operations of the system. We define the semantics of each phase and sub-phase, in the context of Op_{total} (see Section III), the functionality set of our representative system.

- **Solid (P_S) :** If a node has stable predecessor and successor pointers (i.e., the node 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 node can support efficient routing and accommodate up-to-date replica sets, leading to all the upper layer functionalities, e.g., transactional DHT.
- **Liquid :** If a node is on a branch, then it is less strongly connected than those in P_S. However, a node can be on a branch temporarily, e.g., as part of the join protocol. We identify three liquid sub-phases.
  - P_{L1}: If the node is on a branch, but the depth (distance from the core ring) is ≤ 2. The justification of depth of 2 for this sub-phase is based on the evaluation of average branch sizes in [5], that shows that the average branch size of Beernet is ≤ 2, corresponding to the connectivity among nodes on the Internet. So, if a node's depth on a branch is ≤ 2, the operating environment from a node's perspective is the usual one, it might temporarily be pushed on a branch. The node holds a stable finger table and is able to provide transactional functionality to the application.
  - P_{L2}: If the node is on a branch with a depth > 2, but is not the tail of the branch. The finger table at the node holds $> 50\%$ valid fingers. The node is able to support DHT operations, however successful transactions are not guaranteed anymore.
  - P_{L3}: If the node 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 [5]. Also, most of the fingers in the node's finger table are invalid or crashed. From the application's perspective, the node in this sub-phase provides very limited functionality, mostly basic connectivity.
- **Gaseous (P_G) :** If a node is isolated, it is in gaseous phase. As shown in Fig 1, ringlets can be formed due to stress. For example, if there are > 1 nodes per physical machine and the physical network breaks down, then ringlets will be formed by the nodes on the same physical machine. For such scenarios, using the global view we can term the nodes on a ringlet to be in a form of gaseous sub-phase. However, as phase is a node specific local property, a node on a ringlet determines its phase based on its local view, thus its phase decision will be either *solid* or a *liquid* sub-phase.

B. Computation of Phase at Individual Nodes

The phase is a local property of each node. Each node determines its own phase independently, and at a given instant different nodes in the system can be in different phases. The phase of a node should reflect the current stress in its local environment. The phase should be determined with low resource consumption, so that it can be useful even in highly stressful scenarios. Given these constraints, how a node can determine its phase with high confidence, without any global synchronization? In order to determine the current phase at each node by satisfying these constraints, *self-awareness* is crucial for each node. In this section we describe how we have extended each node in our representative system to be self-aware and how a *self-aware* node determines its phase following the semantics described in Section IV-A.

Self-awareness. Knowing its state is not enough for a node to determine phase with high confidence. The node needs to know about its immediate vicinity, and how it is being perceived by the other parts of the system. This combined knowledge provides meaningful context for behavioral decisions and prediction, thus improves accuracy in phase determination. These two types of node's knowledge (i) its internal state, and (ii) how other parts of the system perceive it, can be termed as *private* and *public self-awareness* respectively [12], [13]. A *self-aware* node can determine its phase with high confidence without any global synchronization.

Internal state of a node: The state of a node p is $State_p = \{p, N(p)\}$; where $N(p)$ is the neighborhood (see Section III) of p. For our representative system, $N(p) = \{successor(p), predecessor(p), RT_p\}$, here RT_p is the routing table of p. $N(p)$ is time-dependent due to the dynamics in the overlay. So, each node corrects its internal state, if required, as part of its maintenance strategy.

Achieving public self-awareness: We have extended each node to be self-aware by collecting information about how it is being perceived by the other nodes. A node collects this information as part of its maintenance and also from the messages it receives or routes. The collected knowledge is stored as: $K_{public} = \{Br, Br_{root}, Br_{tail}, Br_{size}, Br_{depth}\}$, where Br, Br_{root}, Br_{tail} are flags denoting existence of a branch and whether the current node is the root or tail of a branch respectively. Br_{size} and Br_{depth} denote the size of the branch and the depth of the node on the branch. In Beernet, each node maintains a list of nodes, called *predList*, that consider the current node as its successor, to do routing on the branches. The list is used to determine the branch size and a node's depth on the branch. Public self-awareness is achieved using both maintenance and protocol messages. For example, consider a node with id u and its predecessor s (Fig 1); u is responsible for the keys $\{t, u\}$. If u receives a message, which is destined to u based on the flag as described in Section III concerning a key q, then u can imply that some nodes of the system perceive it to be responsible of key q. Thus, u can gather the knowledge about the existence of a branch in its local environment and sets the corresponding flag. Similarly, public self-awareness is achieved as part of maintenance also, particularly via proactive maintenance. For example, as part of Periodic Stabilization (PS) [6], each node exchanges periodic messages with its successor to maintain its internal state. Consider a node with virtual id y and predecessor x (see Fig 1). If y receives a PS message from w, which implies w perceives y as its successor and is not aware of or suspecting node x. Thus, y can learn the knowledge of x being on a branch and it itself be the root of a branch, thus it sets the corresponding flags. Similarly, x can also deduce from the absence of periodic maintenance messages from its predecessor about it being pushed as a tail on a branch, thus it sets its flags.

Periodic phase computation. As mentioned in Section II-B, phase at each node is a function of time. So phase determination at a node is periodic. Every T_{phase} time unit, a node computes its current phase. T_{phase} is a configurable parameter. If this time window is too long, several phase transitions, that the node has experienced in between, might be missed to be traced, especially when there is stress in the environment. As a result, the application layer also will not be notified about those, which might affect the performance of the application that invokes phase-driven actions. A better approach is to recompute phase whenever there is a change in a node's internal state or public self-awareness, implementation of which is left as future work.

We can express the phase determination at a node p during a round R, as a function F_{det}. The phase of p at round R is, $Phase_p(R) = F_{det}(State_p(R), K_{public}(R), History_{phase}(p, S))$. Here, $State_p(R)$ is the state of p and $K_{public}(R)$ is the knowledge p has gathered during T_{phase} time unit. $History_{phase}(p, S)$ is a vector which stores the phases of p during the last S rounds. This is required particularly to optimize the decision about the routing table in our context (explained shortly). The function, F_{det}, can be a simple algorithm, such as if-else logic on the fields of the parameters, or a stronger machine learning algorithm. In our implementation, we have used light-weight if-else logic on the flags and values of the fields of three parameters to determine phase at each node. The goal is to demonstrate that using simple logic at each node, without introducing extra resource consumption, the usability and behavior of a distributed application can be significantly improved.

As mentioned in Section III, each node perceives the identifier space to be partitioned into $\log_k(N)$ partitions, where each partition is k times bigger than the previous one. The k-ary routing table of a node contains $\log_k(N)$ fingers to some nodes from each partition. We have assigned weights to the fingers, depending on the partition it belongs to, to compute the invalid/missing fingers. The longest finger to the biggest partition has a maximum weight of $1/k$ and for the next k-times smaller partition, the finger weight

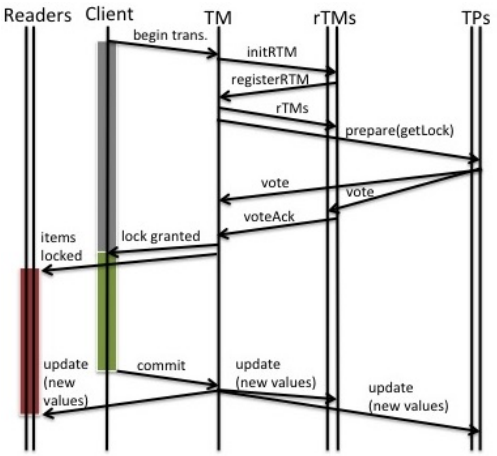


Figure 2: DeTransDraw Transactional Protocol (Eager Paxos with notifications to the readers)

is also reduced proportionally as $1/k^2$, and so on. The motivation for such weighted scheme is that if the longest finger is missing or suspected to be invalid then it affects the routing guarantees more than an invalid finger to the smallest partition, which is in its immediate neighborhood. Consider a scenario: in node p's routing table, one node x is responsible for more than one contiguous partition. As a result p will find missing fingers to some partitions, thus might misjudge the accuracy of its routing table and inaccurately determine its phase during that round. However, it might be the case that the p's routing table is in perfect state as per the global view and the reason for the missing fingers are something different than the effect of stress, e.g., there are only few nodes in the system or skewed mapping of the nodes on the identifier space. Taking these scenarios into account, p must revise its decision about its routing table in subsequent rounds: if during contiguous S rounds, the routing table is in the same inaccurate state, then p can safely assume that this is the optimal state of its routing table as per the global view and re-evaluate its phase decision. p can also proactively verify this, thus optimize its decision, by doing lookups using the ids of the invalid/missing fingers; however also incurring resource consumption. Here, S is a configurable parameter: if S is too low then, a node might misjudge about a scenario, whereas for very high values of S the node will take more time to re-evaluate its phase, also it will consume more memory to store the history.

C. Exposing Phase to Applications

We provide qualitative indication of what system operations are available, i.e., Op_{avail}, by exposing the phase of a node. The phase information is given to the application at each node, using an API [1]. Our API supports push and pull methods: using pull method the application queries the underlying node about its current phase, whereas the push method allows the application to be notified when the underlying node changes phase. With this API, the application can be made reversible, i.e., the application can monitor the qualitative network behavior and change its own behavior accordingly.

V. Phase-Aware Application Design: A Use Case

In order to demonstrate design of a phase-aware application, we have selected a nontrivial use-case application called *DeTransDraw = Decentralized Transactional Drawing (DTD)*, a collaborative drawing tool. The first version of DTD is described in [5]. We extend DTD to integrate phase-aware design decisions in order to improve self-management properties, usability and reversibility of the application.

DTD is a decentralized real-time collaborative vector-based drawing application. It provides a graphical editor with a shared drawing area, and enables coherent collaboration among the users, where each user is graphically notified about the activities of others. DTD is the successor of *TransDraw* [14], which has a client-server architecture. DTD is the decentralized version of TransDraw that runs on a peer-to-peer network. In TransDraw, as the server holds the entire state of the application, it is the single point of failure as well as the source of congestion and scalability issues. In DTD the state is spread across the network and replicated using *Symmetric* Replication [15]. DTD provides load balancing, scalability, and fault tolerance.

DTD is an asynchronous and consistent collaborative application that provides a shared drawing area.DTD uses transactions to hide network latency and improve performance. Transactions allow a user to modify the figures immediately, without waiting for the confirmation of a distributed operation. Due to such optimistic approach (from a user's perspective) a user may lose her modifications if there are concurrent modifications of the same figure by another user, or the user is experiencing intermittent connectivity (i.e., the application has stress in its environment). A conflict resolution mechanism is needed to resolve conflicts, if any, among concurrent transactions. In DTD, the conflicts are resolved by the transaction manager by rolling back one of the conflicting transactions. The probability of conflicts is higher in coherent collaboration. So, we need a pessimistic transactional approach with eager locking, so that the user is notified about any concurrent update earlier, instead of losing all her modifications when trying to commit. A Paxos consensus based commit protocol with optimistic locking [16], where the client performs modifications and then requests locks to commit the changes, hinders the functionality of such applications. So we use an adapted version [5] of Paxos (*Eager Paxos*) to support eager locking with a notification mechanism so that other users are notified on updates or locking of each of the shared items.

Fig 2 illustrates the Eager Paxos transactional protocol as used in DTD. A transaction is done in two steps: i) get the lock and ii) commit. When a user starts modifying objects, the client initiates a transaction by contacting a transaction manager (TM), different for every transaction. Before the TM requests for lock, it registers a set of replicated transaction managers (rTMs) for fault tolerance. If the TM crashes,

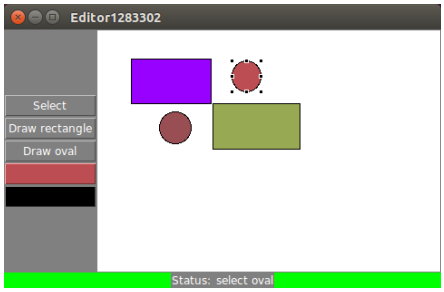


Figure 3: DeTransDraw GUI. The bottom status bar indicates the phase of the node *solid/liquid/gaseous* in green/yellow/red (gray/light-gray/dark-gray) respectively.

the rTMs take over. To get the lock, the TM contacts all transaction participants that store replicas of the object in the transaction. Once the decision (lock granted or rejected) has been made, the client is informed, to prevent users from working on already locked objects. If the locks are granted, new values of corresponding items are committed. DTD merges optimistic and pessimistic approaches to transaction management. It is optimistic as the user does not need to wait for the lock to start working, and it is pessimistic as a transaction first gets the lock and then commits.

Fig 3 shows the graphical user interface of DTD that has four parts: the title bar, the canvas (the shared drawing area), the toolbar and the status bar. The toolbar contains five buttons to set the color and border of the object, to choose drawing operations, and to select objects. When the user selects objects, the client tries to acquire corresponding locks.If the locks are not granted, the objects are unselected and the modifications to these objects are aborted, reverting the objects to their original states. As we can see in Fig 3, the top-right oval is successfully selected by the user. When the user completes modifying the objects, she unselects the objects, the new state of those objects is committed, and object locks are released. The status bar notifies the user about two things. First, *the phase-aware indicator* (color of the status bar) conveys to the user the behavioral prediction of the application (for details see Section V-A); Second, the status bar notifies the user about her current action.

A. Phase-Aware Design Aspects

We extend DTD to integrate phase-aware design decisions in order to enable the application to manage its behavior in stressful environments and achieve self-adaptation and self-optimization. Showing the phase of the underlying node allows DTD to become behaviorally predictable to the users.

Phase-aware behavioral indication. We have integrated *color-based* phase-aware behavioral indicator in the status bar of our application. The indicator can be green or yellow or red, denoting respectively *solid/liquid/gaseous* phase (see Section IV-A) of the underlying node. From the user's perspective, if this indicator is green, she can continue her work without being concerned. If the indicator turns to yellow, the user can expect some functionality disruptions. The red color of this indicator tells the user about her offline state. The indicator in Fig 3 is green, thus assuring the user with full functionality of the application. In this way, the application becomes behaviorally predictable to the user. Such predictability indications allow the users to manage their behaviors and use the application productively, especially when there is stress in the application's environment.

Phase-aware self adaptation. We extend DTD to react to the phase transitions of the underlying node, thus adapting its behavior to the changes in its environment. If the application is notified about a transition from *solid* to *liquid*, the application deduces that the underlying system is not stable anymore, and reacts. It commits all the pending modifications and releases the locks. For example, when the user is doing modifications on one or more selected objects and a transition from *solid* to *liquid* happens, the application commits the changes till that point and releases the locks of corresponding objects, instead of waiting for the user's action to initiate the commit. The goal is to avoid losing all the user's actions at the end, due to intermittent connectivity. However, if the application immediately takes such reactive actions, it might degrade the environment by making the physical links more congested. Also, the user experience might also be affected, e.g., a transient slow-down of the underlying physical link changing the phase for a brief period; for such scenarios reacting immediately will deter the user's work-flow. To avoid these, the application waits for a certain time, T_{React} (a configurable parameter) before reacting. Suppose, the application is notified about the phase transition at T^{th} time unit, it waits till $T+T_{React}$ time unit, if the current phase is still not a stable one, it initiates the reactive actions. Such adaptive actions will enhance user's experience of the application in high-stress environment.

Phase-aware self optimization. We extend DTD to proactively optimize its state based on the recent phase transition history. As there is coherent collaboration among the users, keeping the canvas consistent is important. Suppose, there are three users, A, B, and C, using DTD. B is having intermittent connectivity and misses some updates of other users' activities, making B's canvas inconsistent. In a similar way, A and/or C might miss the notifications about B's activities. To avoid such inconsistencies, the application needs to proactively sync a user's canvas with its global state. However, unnecessary synchronization attempts will only result in resource consumption. The application can initiate such optimization efforts based on the recent phase transitions of the underlying node. If the application is notified about a transition to *gaseous* phase or m times *solid* $\leftrightarrow$ *liquid* phase transitions during a period of T_{opt}, the application initiates a read transaction to sync its canvas with the global state. Here, T_{opt} and m are configurable parameters. We have set $m > 1$ to avoid unnecessary optimization attempts (thus, resource consumption) due to transient transitions to *liquid* phase, because for such cenarios the probability that any update is lost is very low. Thus, based on the phase transitions of the underlying node, the application proactively achieves self optimization.

(a) 4 instances of DeTransDraw (b) User D has crashed (c) User E has joined

Figure 4: DeTransDraw facing Churn. (a) Suppose Users A, B, C and D. (b) D has crashed and is replaced by E. B makes a transient transition to *liquid* phase while doing self-healing, as D was its successor. (c) The system is stable again with full functionality.

VI. Evaluation

We evaluate DTD in stressful environments and analyze the benefits of the phase-aware extensions. We discuss the reversibility of DTD against such environments. Visual demonstrations of our experiments can be found in [17].

We use an overlay of 10 nodes for our experiments. A DTD instance runs on each node. During the steady state of the system, we invoke different stresses, e.g., churn, network congestion, and partitioning. We observe the behavior and available functionalities of the application during the stress and after the stress is revoked. Due to lack of space, we present screen shots of only 4 representative DTD instances.

We simulate churn by terminating one or more DTD instances (i.e., injecting failures at the underlying nodes) and joining new instances. We keep the average number of instances the same. Fig 4 shows the behavior of the DTD instances when one node fails and is replaced by another node. The phase indicators on the status bars (of corresponding successors and predecessors) capture these events by showing a transient transition to the *liquid* phase (in yellow). Due to the reactive maintenance strategy, the concerned nodes correct their states and change to *solid* phase (in green) again after the completion of self healing.

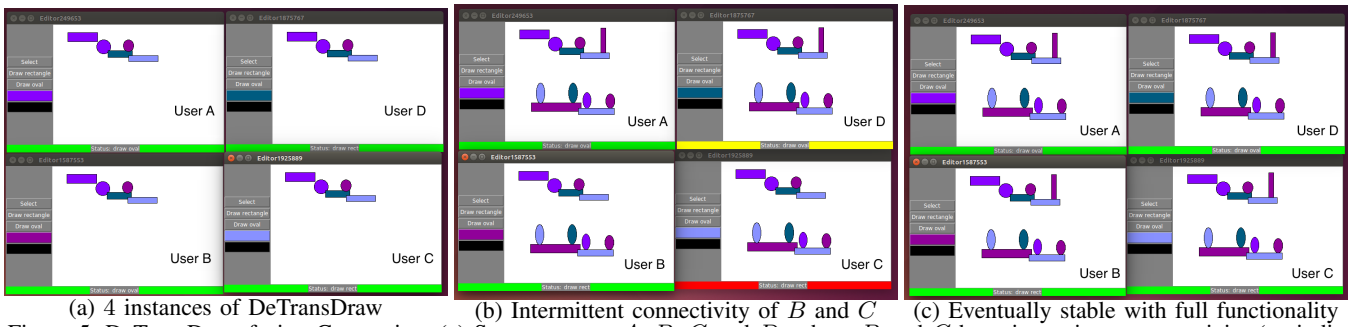
In order to assess the behavior of the application when there is congestion in the physical network, we invoke slow down of node communication. We simulate such scenarios by adding delay (30 sec) on each outgoing message.As the failure detector at each node is adaptive with *Round Trip Time (RTT)* [18], all nodes adapt with the new RTT of a slow node after initial false suspicions, and the system becomes stable with all nodes in *solid* phase. To make the environment more stressful and simulate intermittent connectivity, we trigger periodic slow down of node(s). In our experiments, every minute a slow node slows down by 30 sec for a period of 20 sec, i.e., during this 20 sec period each outgoing message of the node suffers an added delay of 30 sec, followed by a 1 minute period, during which there are no added delays on the outgoing messages. We have invoked 5 rounds of such intermittent connectivity for 2 slow nodes. Fig 5 shows the 3 snapshots of our experiment, where user B and C have intermittent connectivity. D is in *liquid* phase as it is suspecting its successor C, and C is in gaseous phase transiently, because due to its intermittent connectivity, C is also falsely suspecting its predecessor, D and successor, A. However, B is in *solid* phase in Fig 5b, the reason is: B is in a period during which its connectivity has been restored (i.e., no added delay on its outgoing messages). Also, due to facing congestion, both B and C have missed some activities of A and D (see Fig 5b). However, the application eventually recovers all its functionalities at all users, once the system is stable again, as observed in Fig 5c.

Next we experiment with network partitioning. We simulate scenarios where nodes are partitioned away. During the partition, the DTD instances in one partition do not receive updates from the other partitions. As the partition ceases, the application syncs as part of its self-optimization and achieves canvas consistency across users. In our current implementation, the users in one partition lose their modifications done during the partition after the partition is repaired. This was our design choice to retain the highest version of the canvas. This can be improved, by allowing the application to merge the diverging versions accounting to application-level semantics. Fig 6 shows three snapshots of our experiment where a node is partitioned away. As we can see in Fig 6b, during the partition, user A, C and D are in one partition, whereas B is in another partition. With time the canvas across the partitions continues to diverge, i.e., the users in one partition are not aware of the activities of the users in the other partition. After the partition ceases, all DTD instances synchronize with the highest version of the canvas, thus eventually restoring canvas consistency (as in Fig 6c, though B has lost all her modifications done during the partition) and other functionalities.

Reversibility of DeTransDraw. To analyze the reversibility of DTD, first we need to identify its overall functionalities.For DTD, the set of functionalities, $Op_{total} = \{CC, ADD, DEL, SEL, MULTSEL, UPD\}$.

- CC corresponds to consistent canvas across users;
- ADD and DEL correspond to the functionalities of adding and deleting a figure respectively;
- SEL and $MULTSEL$ allow a user to select one and multiple figure(s) on the canvas respectively;
- UPD allows a user to update selected figure(s).

During our experiments, we have assessed reversibility of



(a) 4 instances of DeTransDraw (b) Intermittent connectivity of B and C (c) Eventually stable with full functionality

Figure 5: DeTransDraw facing Congestion. (a) Suppose users A, B, C and D, where B and C have intermittent connectivity (periodic slow-down). (b) The node D suspects its successor C, and makes a transient transition to the *liquid* phase; while C becomes isolated. Node B is in *solid* phase as it is in a period when its connectivity has been restored. Both B and C have missed some activity updates of A and D. (c) As the connectivity of B and C is restored, canvas consistency and all other functionalities are retrieved.

DTD. While experiencing stress, some functionalities may become transiently unavailable, e.g., due to lost updates (as a result of congestion or partitioning in physical network) the canvas consistency gets broken; if a node crashes while holding lock of one or more figure(s) (as a result of selecting those), other nodes are temporarily unable to select and update those particular figure(s), until the lock expires. However, we have established the weak reversibility of our application, i.e., when the stress fades away, the application eventually regains all its functionality.

VII. Related Work

We briefly summarize the relevant work on self awareness in distributed systems. Next, we briefly mention some related works about self adaptation and self optimization in distributed applications. A paradigm shift in system design is explored in [19], [20], from procedural design methodology, where the behavior of the system is pre-programmed, towards self-aware system design, where the system adapts to its context during run-time. In their follow-up work [21], a self-aware computing framework is proposed, which automatically and dynamically schedules actions to satisfy the application's goals. Another approach to build self-managing systems is proposed in [22], based on interacting feedback loops, so that systems can adapt to a wide range of operating conditions and maintain useful functionality. In [23], authors present a high-level methodology for designing the management part of a distributed self-managing application in a distributed manner by partitioning of management functions and orchestration of multiple autonomic managers. The design of a self-aware application to control the behavior of distributed smart cameras is explored in [24]. Based on a utility function, the system decides at run-time how to exchange tracking responsibilities among cameras. A much simpler application of self awareness can be found in [25], in which cognitive radio devices monitor and control their capabilities and also communicate others to achieve efficient communication by negotiating changes in parameters.

Diligent search has failed to uncover any work on phase-aware application design. However, we have found several works on designing self-adaptive and self-optimized distributed systems and applications. The survey in [26] proposes a taxonomy of concerned aspects of adaptation and also presents a landscape of research to identify the research gaps and corresponding challenges. An approach is proposed in [27], following the principle of Separation of Concerns, to systematically develop self-adaptive component-based applications. As per their approach, the adaptation logic is developed separately from the rest and the adaptation policies are interpreted at run-time based on the changes in the environment, detected by a context-awareness service. A predictive stabilization is proposed in [28], as part of Chord, where the nodes of the overlay adapt their maintenance to their environments. Such adaptation improve the performance of the overlay, e.g., low maintenance overhead and high lookup success ratio. The work in [29] develops a mechanism to optimize the resource consumption between nodes by transferring services to other nodes. In our work, the context-awareness is provided to the application by the underlying system, in the form of phases, based on which the application can trigger its adaptation and optimization policies, thus can be seen as complementary to these works.

VIII. Conclusion

In order to take full advantage of edge computing, large-scale applications must be designed considering the high stress inherent in such operating environments. In this paper, we have proposed an approach for designing applications for such environments. We have built our application on top of a *Reversible* and *Phase-Aware* Structured Overlay Network, hosting a transactional DHT. We have applied and expanded the concept of *Reversibility* in the context of application design. We extend each node of the underlying system to be *self-aware* and demonstrate how a self-aware node can determine its phase without any global synchronization. We have applied the concept of phase to approximate the available operations in our system, and have demonstrated how this information can be made available to the application layer by exposing the phase of the underlying node. We have described phase-aware design of one use-case application, namely a collaborative drawing tool. Using this use-case application, we exhibit, based on the phase and

(a) 4 instances of DeTransDraw (b) User B is partitioned-away (c) After the Network Partition ceases

Figure 6: DeTransDraw facing Network Partition. (a) Suppose users A, B, C and D, where B is partitioned away. (b) During the partition the canvas across the partitions diverges with time. (c) When the partition ceases, the application, being self-adaptive and self-optimized, syncs based on the observed phase transitions, and achieves the canvas consistency and full functionality.

phase transitions of the underlying node, how an application can achieve reversibility in the application-level semantics. Furthermore, the application can have improved behavior from the viewpoint of the user, i.e.: have self adaptation, self optimization, and, be behaviorally predictable to the user in high-stress operating environments.

In our future work, we intend to investigate other design aspects based on the phase concept that can further enhance the application's behavior. Also, we intend to do large-scale experiments in real-life stressful environments.

ACKNOWLEDGMENT

This research is supported by the SyncFree project funded by the European Commission in FP7 under Grant Agreement No. 609551, and by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the EACEA of the European Commission under FPA $2012 - 0030$.

REFERENCES

[1] R. R. Paul, P. Van Roy, and V. Vlassov, "Reversible phase transitions in a structured overlay network with churn," in *Proc. NETYS*, 2016.

[2] ——, "Interaction between network partitioning and churn in a self-healing structured overlay network," in *ICPADS*, 2015.

[3] Wikipedia. (2016) Phase (matter). https://en.wikipedia.org/wiki/Phase_(matter).

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, 1974.

[5] B. Mejías, "Beernet: A relaxed approach to the design of scalable systems with self-managing behaviour and transactional robust storage," Ph.D. dissertation, UCL, Belgium, 2010.

[6] 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.

[7] K. Gummadi *et al.*, "The impact of DHT routing geometry on resilience and proximity," in *Proc. ACM SIGCOMM*, 2003.

[8] K. Aberer *et al.*, "The essence of P2P: a reference architecture for overlay networks," in *Proc. P2P*, 2005.

[9] 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.

[10] M. Jelasity and O. Babaoglu, "T-Man: Gossip-based overlay topology management," in *Proc. ESOA*, 2005.

[11] C. Meiklejohn and P. Van Roy, "Lasp: A language for distributed, coordination-free programming," in *PPDP*, 2015.

[12] M. Mitchell, "Self-awareness and control in decentralized systems," *Metacognition in Computation*, pp. 80–85, 2005.

[13] P. R. Lewis *et al.*, "A survey of self-awareness and its application in computing systems," in *Proc. SASOW*, 2011.

[14] D. Grolaux, "Editeur graphique réparti basé sur un modéle transactionnel," Master's thesis, UCL, Belgium, 1998.

[15] A. Ghodsi, "Distributed k-ary system: Algorithms for distributed hash tables," Ph.D. dissertation, KTH, Sweden, 2006.

[16] M. Moser and S. Haridi, "Atomic commitment in transactional DHTs," in *Proc. CoreGRID*, 2007.

[17] R. R. Paul and J. Melchior. (2016) Demo phases. https://www.youtube.com/playlist?list=PLAe-bnT0VkXjFxkmph3mjkpJ-4fBQXN05.

[18] R. R. Paul, P. Van Roy, and V. Vlassov, "An empirical study of the global behavior of a structured overlay network," in *Proc. P2P*, 2014.

[19] A. Agarwal, J. Miller, J. Eastep, D. Wentziaff, and H. Kasture, "Self-aware computing," MIT, Tech. Rep., 2009.

[20] A. Agarwal and B. Harrod, "Organic computing," MIT and DARPA, Tech. Rep., 2006.

[21] H. Hoffman, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A General and Extensible Framework for Self-Aware Computing," MIT, Tech. Rep., 2011.

[22] P. Van Roy, S. Haridi, and A. Reinefeld, "Designing robust and adaptive distributed systems with weakly interacting feedback structures," UCL, Belgium, Tech. Rep., 2011.

[23] A. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, "A design methodology for self-management in distributed environments," in *IEEE CSE*, 2009.

[24] L. Esterle, P. R. Lewis, M. Bogdanski, B. Rinner, and X. Yao, "A socio-economic approach to online vision graph generation and handover in distributed smart camera networks," in *Proc. ICDSC*, 2011.

[25] J. Wang, D. Brady, K. Baclawski, M. Kokar, and L. Lechowicz, "The use of ontologies for the self-awareness of the communication nodes," in *Proc. SDR*, 2003.

[26] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM TAAAS*, 2009.

[27] P. C. David and T. Ledoux, "Towards a framework for self-adaptive component-based applications," in *Proc. DAIS*, 2003.

[28] R. Kaur, A. L. Sangal, and K. Kumar, "Modeling and simulation of adaptive neuro-fuzzy based intelligent system for predictive stabilization in structured overlay networks," *Engineering Science and Technology, an International Journal*, vol. 20, no. 1, pp. 310 – 320, 2017.

[29] W. Trumler, A. Pietzowski, B. Satzger, and T. Ungerer, "Adaptive self-optimization in distributed dynamic environments," in *Proc. SASO*, 2007.