RepFD - Using reputation systems to detect failures in large dynamic networks

Maxime Véron*, Olivier Marin*†, Sébastien Monnet*, Pierre Sens*
* Sorbonne Universités, UPMC Univ Paris 06, CNRS,
INRIA, LIP6 UMR 7606, 4 place Jussieu 75005 Paris. {Firstname.Lastname}@lip6.fr
†NYU Shanghai. ogm2@nyu.edu

***Abstract*—Failure detection is a crucial service for dependable distributed systems. Traditional failure detector implementations usually target homogeneous and static configurations, as their performance relies heavily on the connectivity of each network node. In this paper we propose a new approach towards the implementation of failure detectors for large and dynamic networks: we study reputation systems as a means to detect failures. The reputation mechanism allows efficient node cooperation via the sharing of views about other nodes. Our experimental results show that a simple prototype of a reputation-based detection service performs better than other known adaptive failure detectors, with improved flexibility. It can thus be used in a dynamic environment with a large and variable number of nodes.**

Keywords*—*Failure detection ; Reputation Systems ; Large scale distributed systems

I. Introduction

Distributed systems should provide reliable and continuous services despite the failures of some of their components. A classical way for a distributed system to tolerate failures is to detect them and then to recover. It is now well recognized that the dominant factor in system unavailability lies in the failure detection phase [1]. As a consequence, failure detection plays a central role in the engineering of such systems. Chandra and Toueg introduced in [2] the notion of unreliable failure detector (FD). An FD is an oracle which provides information about process crashes. It is unreliable as it can make some mistakes for a while; for instance, some live nodes can be considered as having crashed. FDs are used in a wide variety of settings, such as network communication and group membership protocols, computer cluster management and distributed storage systems. Numerous implementations of FDs have been proposed, where each node monitors the state of the others. However, most FD implementations have two severe limitations:

- they consider all the nodes in a same way, there is no distinction between well and bad behaved nodes;
- local oracles gather information from the other nodes without any coordination [3], [4], [5].

In stable and homogeneous configurations such as clusters, where nodes of a same type are linked through low latency networks and subject to crash failures at the same rate, these limitations have a low impact on the quality of the failure detection. However, in large and dynamic systems such as gaming platforms or large cloud infrastructures, nodes are very different: some nodes (eg. Server) are powerful and connected to the network with a high speed link whereas some others have a limited power and slow connections. Taking into account such differences is essential for the quality of the detection. Furthermore, in such dynamic environments sharing information on the state of the nodes could greatly increase the global view of the distributed system. If one node has a good connection to the other ones, it can share its view to slowly connected nodes and thus prevent wrong views about failures.

In this paper, we propose a new collaborative failure detector which exploits information about the behavior of nodes to increase its detection quality both in terms of detection time (completeness) and mistake avoidance (accuracy). To classify the behavior of nodes we rely on a reputation service where nodes periodically exchange heartbeat messages. The reputation of a node dynamically increases if it sends its heartbeat on time, and decreases if some heartbeats get lost or arrive after the expected dates.

We conducted an extensive evaluation of our failure detection on distributed configurations using real traces to inject failures and message losses. We show that our detector outperforms well-known implementations [4], [6]: it provides a better accuracy while keeping short detection times, especially when the network is subject to message losses.

The rest of the paper is organized as follows. Section II presents the reputation system we use to implement our failure detection service and details the detector implementation. Section III describes two standard failure detector implementations we then compare to our solution in the performance evaluation of Section IV. Finally, Section V explores related work and Section VI concludes the paper.

II. Detecting failures with a reputation system

Our solution uses a distributed reputation system to detect failures. A reputation system [7] aims to collect and compute feedback about node behaviors. Feedback is subjective and obtained from past interactions between nodes, yet gathering feedback about all the interactions associated with one node produces a rather accurate representation of its behavior. In our case, the reputation system focuses on behaviors that fall within the scope of a given failure model.

The reputation system we present in this section is basic and aims to reproduce the qualities of a good reputation system according to [8]: fast convergence, precise notation of nodes, resistance to malicious nodes, small overhead, scalability, and adaptivity to peer dynamics. Our reputation system can be used for a wide variety of middlewares and services. In a previous work, we describe in details and use this reputation system to

support another kind of application, namely cheat detection in massively multiplayer online games [9].

A. Assessment of the reputation

Every node stores a local estimation of the *reputation* associated with every node in the network. In our system, a reputation value belongs to $[0, 1000[$. Value 0 represents a node which never delivers its service correctly, whereas the reputation value of a very trustworthy node tends to 1000. Initially, a node with no known history in the network has its *reputation* value set to 0.

A reputation assessment primarily consists in comparing inputs from neighborhood nodes with an expected behavior. Applying this scheme to failure detection is simple: if a node sends its heartbeat in a timely manner its reputation value increases, otherwise it decreases according to a reputation *punishment*. We call this primary assessment a *direct* assessment; it is in all ways similar to traditional failure detection. To improve the detection in dynamic environments, we combine the *direct* assessment with an *indirect* assessment.

The *indirect* reputation assessment of a node A by another node B relies on three pieces of information: (i) the current reputation value that B associates with A, (ii) the evolution of the behavior of A as perceived by B, (iii) and the recent opinions that B or other nodes may express about A.

Upon receiving fresh data about the behavior of a node, reputation systems such as TrustGuard [10] use a *PID (proportional-integral-derivative)* formula on these three pieces of information to compute a new reputation value. The principle of a PID formula is to carry out a weighted sum of the local reputation value at $t-1$, of the integral of the local reputation values since the system startup, and of the differential with newly received reputation values. Let $R(t)$ denote the reputation value of node n at time t:

$$R(t+1) = \alpha * R(t) + \beta * \tfrac{1}{t} * \int_0^t R(x)\,dx + \gamma * \tfrac{d}{dt} R(t)$$

Our reputation system simplifies this computational model to facilitate its implementation. This simplification, as detailed in [9], transforms the full model into an arithmetic progression. On any node in the network, the local computation of the reputation $R_m(t+1)$ associated with node m in our implementation thus breaks down to:

$$PID(a,m) = \alpha * R_m(t) + \beta * \tfrac{I_m(t-1)+a}{2} + \gamma * (a - R_m(t))$$
$$R_m(t+1) = PID(a,m)$$

where $I(t)$ is the approximate value of the integral at time t and a is the assessment of m at time t. a can be a reward or a punishment for m, but it can also be a reputation value for m received from another node in the network. Thus we combine the local assessment with views from other nodes of the network to compute R_m. This allows us to build up a global view of the system on each node without overhead.

B. Parameters associated with our reputation system

Several parameters associated with our reputation system allow to adapt it to the requirements of the application. Fine tuning the values of these parameters requires a test phase on the target network. Subsection IV-A includes a description of our benchmarking methodology to adjust the settings of our reputation-based failure detector. The present Subsection gives general pointers for the parameterization of the reputation system.

The first obvious set of parameters α, β, γ characterizes every reputation assessment. A high value for α will confer a greater importance to past reputation values stored locally. This is useful in systems where close neighbors behave erratically. Parameter β focuses on the history of the behavior of a node. Systems with a high value for β slow down reputation decreases induced by sporadic changes in the network such as bursts of message losses. Finally, parameter γ reflects the direct impact of a new opinion on the local assessment. A high value for γ implies that the local reputation value of a node will be more sensitive to new values expressed locally or by other nodes. In the context of fault detection, we believe the system should focus on its tolerance to jitter and to message losses. As the main component of the formula for this purpose, β ought to be set to a high value.

We determine whether a node acts correctly by enforcing a global threshold T on reputation values: a node with a reputation greater than T is considered as correct. T must be set to the best trade-off between the accuracy and the efficiency of the failure detection, and its value is highly dependent of the quality of network links. If node connections incur high jitter and/or high message loss rates, a high value of T will induce many false failure detections. On the other hand, low values of T could dramatically increase the detection time. As this threshold is a crucial metric for determining faulty nodes, we also describe how we fix its value in Subsection IV-A.

There are other, more secondary parameters with respect to detection. Values υ (up) and δ (down) correspond respectively to rewards and punishments. Also, a $decay$ factor affects all locally stored reputation values upon a $reassessment$ timeout. This strategy aims to force nodes to reassess reputations for nodes with which they have no direct interaction for some time. High values for the reassessment frequency and the decay factor reduce the detection time, but increase the rate of false detections.

C. Our reputation based failure detector

Algorithm 1 presents our reputation-based failure detector implementation. In task T1, every node sends its neighbors a heartbeat message every Δ_H. Every heartbeat encapsulates the reputation information the node cares to propagate. When a node p receives a heartbeat message from a node q (task T2), it computes a new reputation $R_q(t+1)$ which rewards q with a maximum update value $a = \upsilon$ (line 11). Then for each reputation information included in the message, p updates the reputation of the corresponding node (lines 17–18). For an eventual decrease of the reputation of all known faulty nodes, a reputation reassessment occurs every HC heartbeats: p then applies a $decay$ factor to all locally known reputation values (lines 14–15). To boost local detection, p punishes nodes which haven't sent heartbeats for two periods (task T3). We used this value to be responsive whilst being resistant to small jitters. Finally, task T4 handles fault detection requests from

```
Task T1 [HeartBeat / Reputation propagation]
Every Δ_H

    For each known node n Do
        Insert REPUTATION(R_n, n) in heartbeat
    Send HEARTBEAT to each neighbor

Task T2 [Heartbeat reception]
Upon reception of HEARTBEAT from q

    R_q ← PID(v, q)
    If HCounter = HC Then
        { Refreshment of reputation of all known nodes }
        For each known node n Do
            R_n ← R_n − decay
        Hcounter ← 0
    For each REPUTATION(R_n, n) in heartbeat Do
        R_n ← PID(R_n, n)
    HCounter ← HCounter + 1

Task T3 [Punishment]
Every Δ_H * 2

    For each known node n Do
        If !received(HEARTBEAT, n) Then
            R_n ← PID(δ, n)

Task T4 [Decision]
Upon asking a detection for node n

    If R_n > T Then
        return NodeCorrect
    Else
        return NodeFaulty
```

Algorithm 1: Reputation based failure detector algorithm

the application layer. Our detector considers a node is correct if its reputation value is greater than a given threshold T.

D. Scalability of our reputation-based detector

In order to scale up to a large number of nodes, our reputation system imposes a network topology that avoids all-to-all communication. We conceived a simple algorithm (Algorithm 2) for the random generation of connected digraphs where every *common node* has an average degree of 3. In order to reproduce the skewness of connectivity in large scale overlays, our algorithm also randomly designates *super nodes* that possess direct links towards a third of the network. A node has a probability P_{super} of being a *super node*. Our algorithm uses a coordinator node that builds a static network topology for all connected nodes. Initially each node sends its identifier to the coordinator (Task T1). The coordinator then sends each node the list of its neighbors in the overlay (Task 3). In the algorithm, $Random()$ generates a random number in the interval $[0, 1]$ and $Subset(S, c)$ returns a random subset of set S with a cardinality equal to c.

As we showed in [9], our reputation system scales extremely well with respect to the number of nodes involved. Even though the solution may sound resource intensive, it only uses 160 bytes per second per node in a system with 30000 nodes running the reputation system. This is due to the small size of the reputation data that gets exchanged.

In the performance evaluation of Section IV, we limited the

```
P_super ← 0.1
min_neighbors ← 2
max_neighbors ← network_size/3
network ← ∅

Task T1 [Node Initialisation]

    Send ID(p_i) to Coordinator

Task T2 [Neighbors reception]
Upon reception of NEIGHBORS(list) from Coordinator

    neighbors ← list

Task T3 [ID reception]
Upon reception of ID(p_j)

    network ← network ∪ p_j
    If |network| = network_size Then
        For each node n in network Do
            If Random() > P_super Then
                list ← Subset(network, max_neighbors)
            Else
                list ← Subset(network, min_neighbors)
            Send NEIGHBORS(list) to n
```

Algorithm 2: Overlay generation

size of the network to 10 nodes to allow for a large number of experiments over varying configurations. In particular, Subsection IV-H studies and compares the bandwidth consumption of the three approaches in various experimental scenarios. However we also ran an experiment involving 250 nodes to check how our reputation-based failure detector behaves over a larger network. We observed that, regardless of the network size, the bandwidth consumption of our detector remains steady. Every node that participates to our reputation system consumes an average bandwidth of 144 Bytes per second, with almost no deviation. We consider that this value is very low and well within the capacity of most Internet connections nowadays.

III. Comparison with other failure detectors

In order to assess our reputation oriented approach, we compare it with two other oracle-based failure detectors: Bertier [4] and Swim [6]. These failure detectors constitute references of efficient and well-known failure detectors. They are briefly presented in this section, along with a first assessment of each detector's impact on the network.

A. Bertier's failure detector

Bertier's failure detector combines one of Chen's estimations [11] for the arrival time of heartbeat messages and a dynamic safety margin based on Jacobson's algorithm [12].

Chen's estimation computes the expected arrival time EA of heartbeat messages, and adds a constant safety margin α to avoid false detections caused by transmission delay or processor overload. EA results from adjusting the theoretical arrival time of the next heartbeat with the average jitter incurred upon the n latest heartbeat receptions. The value of α requires a calculation with respect to QoS requirements prior

to starting the system; it can not account for radical alterations of the network behaviour.

Bertier's failure detector solves this issue by using Jacobson's algorithm to compute a dynamic safety margin. Jacobson's estimation assumes little knowledge about the system model; it is used in TCP to estimate the delay after which the transceiver retransmits its last message. This estimation relies on the error incurred upon reception with respect to the estimated arrival time, and includes user-defined parameters to weight the result.

B. Swim's failure detector

Swim [6] relies on a ping approach. An initiator node invites k other nodes to form a group, pings them and waits for their replies. If a node does not reply in time, the initiator then judges this node as suspicious and asks the other group members to check the potentially faulty node. If this node remains silent after three consecutive pings from all group members, the detection is confirmed and the node is finally considered as incorrect.

Swim's failure detector allows a fair comparison in that, instead of requiring an *all to all* communication, it shares the same type of network footprint as a reputation system.

C. Communication complexity

The three algorithms we compare have different impacts on the network.

1) Bertier: Bertier's failure detector induces all to all communication. Upon every period, every node sends a heartbeat to every other node: their message complexity is n^2 messages per period.

2) Swim: The failure detector in Swim has two operational modes: a standard one where the initiator assesses its group of K nodes, and a degraded mode where all K nodes assess a non-responsive node. This derives two message complexities: (i) $k * n$ messages per period in standard mode, and (ii) $k * n^2$ in the worst case of the degraded mode.

3) Our reputation-based detector: Reputation systems usually incur a high communication complexity, so we reduced our network footprint as much as possible. Nodes send periodic heartbeats to their neighbors only, and propagate reputation data by piggybacking it on the heartbeats. To reduce network consumption even further, nodes only emit reputation values that differ significantly from the last emitted value. This produces a difference of behavior between good and bad nodes as we will save a lot of bandwidth on good nodes reputation propagation. In our experiments, setting the minimal variation before emission to 50 on a $0 - 5000$ scale reduced message size by 50%, with no effect whatsoever on the quality of the detection.

Let d be the average degree of nodes in the system: the complexity of our detector is $d * n$ messages per period.

IV. Performance evaluation

This section presents an evaluation of our approach. We assess its performance and compare it with both Bertier's [4] and Swim's [6] state of the art failure detectors. Throughout this section we use *BertierFD*, *SwimFD*, and *RepFD* to refer respectively to Bertier's, Swim's, and our reputation-based failure detector.

A. Experimental settings

a) Application settings: All scenarios are run on ten nodes for a duration of five minutes, with a heartbeat/ping period of one second. Each experiment is run fifty times. The standard deviation consistently remained very low: under 4% for all our experiments. Hence we chose not to include it in our figures.

b) Network settings: We ran our experiments on a cluster made out of dual-Intel Xeon X5690 running at 3.47Ghz and equipped with 143GB of RAM. Since our cluster incurs near zero latency, we injected a 59ms latency for each message to reproduce typical user broadband connections. This value comes from our study of user experience that capitalizes on statistics published by a very popular online game [13]. Our code uses the UDP protocol for message exchanges, thus preventing detections caused by connection closures.

BertierFD relies on all to all communication so we organized the network in a clique. For SwimFD, we fixed the membership set to involve all ten nodes. Finally, we generated the topology for our reputation system at random upon every experiment by using the algorithm described in Subsection II-D. The resulting graph is connected and the degree of every node is greater than or equal to two. This topology limits communications overhead while ensuring the liveness of the propagation of reputation information among nodes that generate the global view.

c) Failure detector settings: We set $\Delta_H = 1s$ for all the assessed failure detectors. BertierFD and RepFD send heartbeat messages every second. SwimFD also sends ping messages every second in order to obtain coherent detection times.

BertierFD requires an additional setting: the initial detection time beyond which a missing heartbeat determines its sender as suspicious. We followed the original implementation and set this value to Δ_H. All other parameters were also set to the values advocated in [4].

RepFD relies on a reputation assessment which requires some parameterization. The essential parameters of this assessment are the weights α, β, and γ of the PID formula that computes reputation values, as well as the threshold T that distinguishes suspicious nodes from correct ones according to their computed reputation value. To adjust the values for these parameters, we first ran brute force simulations on top of the Peersim simulator [14]. We set the reward value υ to 1000 (maximum reputation value), the punishment value δ to 0 (minimal reputation value), $decay$ value to 50, HC to the size of the network, and then ran simulations where α, β, and γ varied from 0 to 1 and T varied from 400 to 800. Our detector achieved its best quality of detection with the following settings: $\alpha = 0$, $\beta = 0.8$, $\gamma = 0.2$, and $T = 700$.

To reach a better understanding of why these settings work well, we experimented further on top of our ten-node cluster. We set threshold T to 700 and measured both the accuracy

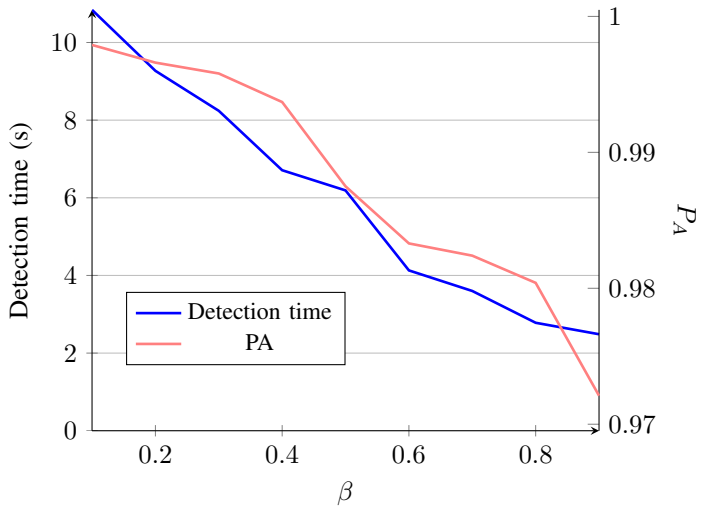


Fig. 1: Impact of beta and gamma on the quality of the detection

and the detection time of our detector with varying values for α, β, and γ. We started with a separate study of each weight. To do so, we initially incremented the value of a single weight by 0.1 between 0 and 1, and set the other two weights equal to $\frac{1-(studied\_parameter)}{2}$. Conversely to β and γ in these experiments, we observed that no value of α leads to a good quality of detection. $\alpha > 0.25$ prevents any detection, while values below 0.25 induce detection times of at least 3.74 seconds. We will show in the following experiments that this is a poor detection time, but one can reach this conclusion intuitively as it represents almost four times the period between heartbeats. Hence the first conclusion we can draw from our study is that the local view of past detections bears too much influence on future detections and should be discarded entirely.

We therefore focused our study on β and γ: we incremented the value of γ by 0.1 between 0 and 1, and set $\beta = 1 - \gamma$. Figure 1 plots the detection time and the accuracy for all values of β in this last series of experiments. Introduced in [11], query accuracy probability (noted P_A) reflects the probability that a failure detector's output is correct at any random time. These results confirm those reached through our simulations: beyond $\beta = 0.8$ the detection time will not decrease much further, but the accuracy starts shooting down. Our understanding is that detection assessments from distant nodes, included in γ, actually slow down the local detection and help avoid mistakes. Meanwhile, β has a positive influence on detection time as it reflects the consistency between the latest local detections and those of neighbor nodes.

This extensive study leads to the following settings for our reputation-based detector throughout our experiments: $\alpha = 0$, $\beta = 0.8$, $\gamma = 0.2$, and $T = 700$.

B. Measuring the accuracy of multiple detectors

In order to measure the accuracy of the detection, we introduce a metric we call *global node correctness*. The *global node correctness* of a node n is the average number of correct nodes that deemed n correct. A value of 1.0 means that all nodes consider n as correct.

First we define a $correctness$ value for any node n of the network as:

$$correctness(n, p) = \begin{cases} 1, & \text{if } n \text{ is deemed correct by p} \\ 0, & \text{otherwise} \end{cases}$$

With our reputation system, a node n with reputation R_n is deemed correct if $R_n > T$.

To calculate $global\ node\ correctness$ at any given moment, we compute the $global\ node\ correctness$ of a node n as the average correctness associated with n throughout the network.

$$global\ node\ correctness(n) = average(\ correctness(n, p), \forall\ p \in network)$$

For example with a perfect detector all nodes will detect incorrect nodes as incorrect, therefore the average $global\ node\ correctness$ of the incorrect nodes will be 0.

C. False positives in the absence of failures

We first measure and compare the accuracy of the studied failure detectors by testing for false positives in the absence of failures.

BertierFD initializes its heartbeat detection period to Δ_H, and then requires some time to estimate the real RTT of nodes which includes the latency. It takes approximately 14 seconds to converge but behaves extremely well once the system stabilizes : it does not produce a single false positive. The resulting plot, a straight line, is therefore uninteresting and we decided not to include the figure in this paper.

SwimFD also converges very fast: its stabilization time is similar to that of BertierFD, around 14 seconds. However, even in the absence of failures, SwimFD does not handle jitter of connection very well. As shown in Figure 2, small changes of latency quickly drive nodes into SwimFD's list of suspicion. A suspicion does not directly translate to a detection for SwimFD, though: an overdue heartbeat launches the degraded mode where nodes in the same group exchange their views on the suspected nodes. Overall SwimFD introduces a lot of false suspicions and thus generates significant network usage overhead, but it does not produce false detections.

RepFD is noticeably slower at converging: it takes about 30 seconds to stabilize. This is the time required by the underlying reputation system to build its view of the network. Upon stabilization, RepFD also behaves well: it produces no false detection in the absence of failures. Figure 3 plots the average reputation value over time. RepFD initializes reputation values at 0, but starts storing them after the first call to *PID* function: hence the first reputation value represented on the graph is higher than 600. Since a correct behavior systematically entails a reward υ for the observed node, the reputation of correct nodes quickly converges towards the highest possible value and will never decrease below the 700 threshold value.

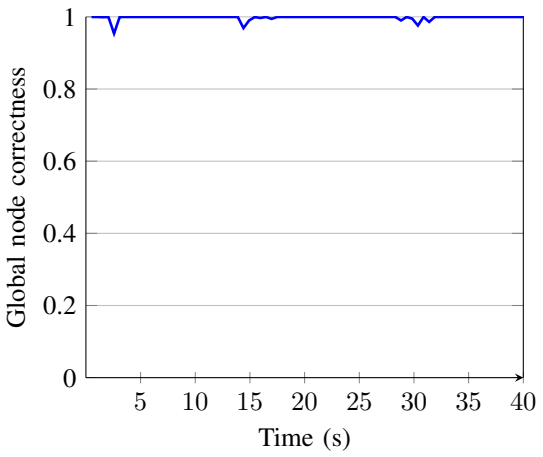


Fig. 2: Accuracy of SwimFD in a failure free environment

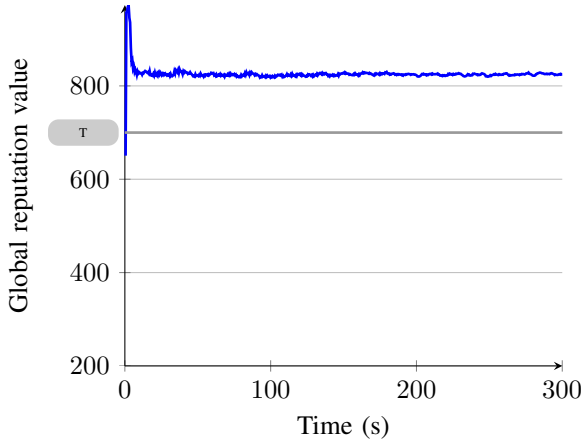


Fig. 3: Accuracy of RepFD in a failure free environment

D. Permanent crash failures

After analyzing the behavior of the detectors in a failure free environment, we introduce silent crash failures. At the initialization of the experiment, the system randomly designates two nodes as faulty. Nodes programmed for incorrect behaviors crash silently and never recover. An incorrect node implements the silent crash by shutting down its network connections after the full stabilization time of the studied detectors. We therefor set the crash time to 50 seconds in order to ensure that all systems have fully converged.

The detection time corresponds to the time elapsed between the moment the failure occurs and the moment all correct nodes detect the failure. As such it is relative to the length of the heartbeat period but, in our case, setting the same heartbeat period for all three detectors offers a fair comparison.

Table 4 displays the average detection times exhibited by the detectors. All three detectors react within the same timeframe. SwimFD may seem faster: its displayed detection time is roughly 20% shorter than those of BertierFD and RepFD. However, SwimFD's detection time corresponds to the time elapsed from the moment of the crash to the moment all correct nodes have pushed the faulty node in their list of suspicion. The true detection requires three more RTTs among the nodes of the detection group.

	BertierFD	SwimFD	RepFD
Detection time (s)	2.48	1.97	2.578

Fig. 4: Detection time of a permanent crash failures

We also study the accuracy of the detection for each detector. For this purpose, we measure the *global node correctness* of correct nodes and that of faulty nodes separately over time. Figures 6, 7, and 8 represent our results for BertierFD, SwimFD, and RepFD respectively. All three detectors behave similarly towards faulty nodes: once they start suspecting a permanent failure, they will not reconsider. SwimFD does not handle correct nodes as well as BertierFD and RepFD, however: it keeps suspecting them intermittently.

The overall conclusion of this set of experiments is that, even though SwimFD takes a little less time to reach a decision towards suspicion, it pays a heavy price in terms of accuracy. RepFD and BertierFD are much more stable in their estimation of correctness. It is important to point out that BertierFD achieves this over an all to all communication protocol, whereas a connected graph suffices for our failure detector.

E. Crash/recovery failures

The following set of experiments evaluates the behavior of the studied failure detectors in a more complex environment, where faulty nodes stop receiving/sending messages for a given amount of time and then resume communications. In our crash/recovery scenario, faulty nodes cease their network interactions every 30 seconds for a duration of 30 seconds. Please note that this scenario is not common in the literature about failure detectors; most studies focus on permanent crashes. However, crash/recovery failures are far more representative of the large and dynamic systems we target.

We first study the detection time in this series of experiments, and collect measures taken after each detector's respective stabilization time. Table 5 displays the average detection times for all three detectors.

	BertierFD	SwimFD	RepFD
Detection time (s)	3.19	2.6020	2.5776

Fig. 5: Average detection times in crash/recovery environments

Our first observation is that, in this scenario too, all three detectors spot failures within the same timeframe. Yet upon closer inspection, it appears that introducing recovery does affect the detection times of BertierFD and of SwimFD: compared to the permanent crash scenario, they increase by 28% and by 32% respectively. Conversely, the introduction of recoveries does not impact the average detection time of RepFD. The increase is not surprising in the case of SwimFD. SwimFD does not rely on a history of past detections but on a group decision. Given its lower accuracy and the fact that it generates a considerable amount of network overhead upon suspicion, its average detection time converges towards

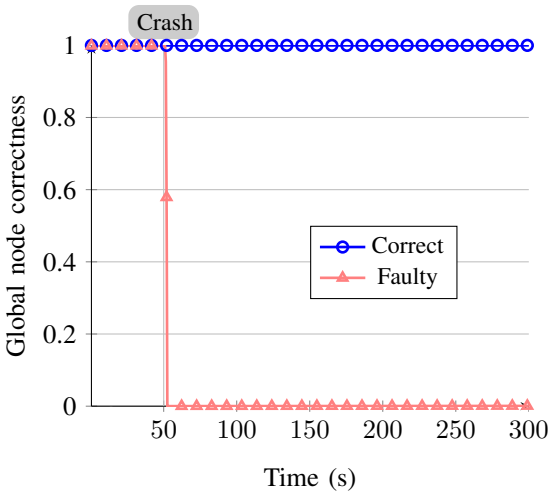


Fig. 6: Accuracy of BertierFD with respect to fail silent failures

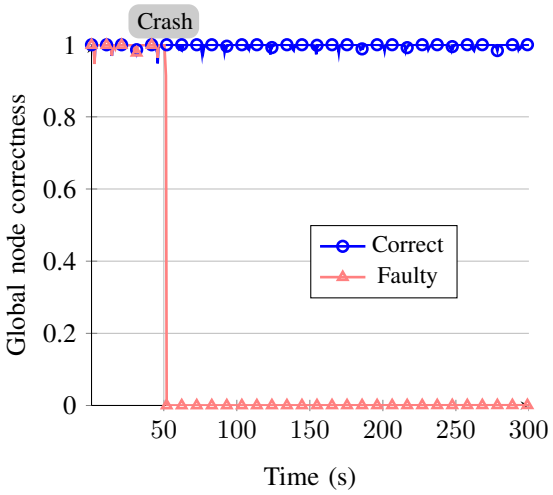
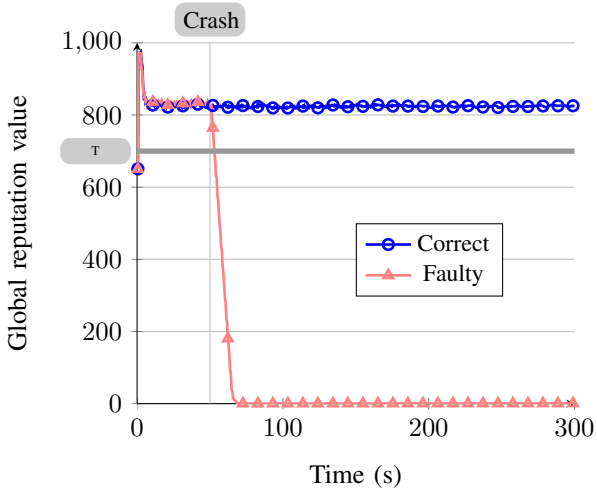


Fig. 7: Accuracy of SwimFD with respect to fail silent failures

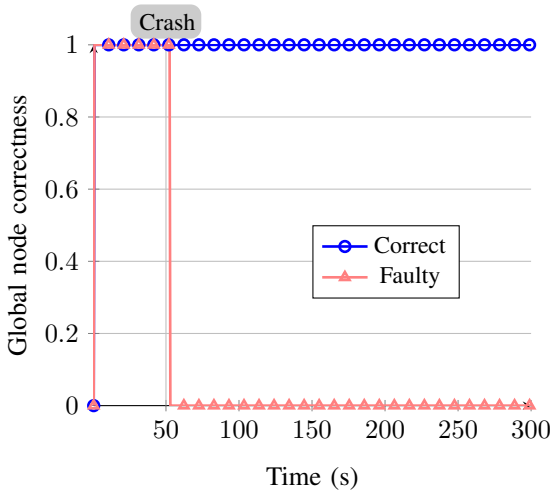
that of the other detectors when repeated occurrences of multiple crash/recoveries complicate the detection. The more complicated it is to reach a common decision, the more time it takes for SwimFD detectors to reach it. BertierFD and RepFD both rely on past detections, so in light of the permanent crash experiment results we were expecting their detection times to be similar again. However we were underestimating the sensitiveness of Bertier's RTT estimation to latency variations. An analysis of our logs showed indeed that our cooperative reputation assessment adjusts much faster to crash/recovery than Bertier's RTT estimation computed single-handedly on every node.

We then study the accuracy of each studied detector in our crash/recovery scenario. As expected, SwimFD has a hard time avoiding mistakes in its assessment of both correct and faulty nodes. Figure 10 illustrates this situation: both graphs show that all nodes incur a lot of suspicion, even nodes that behave correctly throughout the experiment. BertierFD is far more stable (Figure 9): there are no haphazard mistakes once a node is considered either correct or suspect. But changes of state of faulty nodes lead to some degree of hesitation for BertierFD: the points where faulty nodes stop/resume their communications do not appear as sharp angles on the associated graph. This hesitation comes from the RTT reevaluation every time a change of state occurs. Our reputation-based detector produces the results shown in Figure 11b: no jitter and clear-cut angles. This shows that RepFD responds very well to the crash/recovery scenario. Figure 11a explains why by plotting the evolution of the reputation values used to determine correctness. The uncertainty associated with the assessment translates to the oscillation of the reputation values, while the threshold enforces both mistake avoidance and a quick discrimination between correct and faulty nodes.

Another interesting point of our approach is that, while detectors gather reputation values, they also build a global view of the network. Therefore, even temporarily isolated nodes will still have a global view of the system while the network link is down. This provides meaningful data for identifying points of failure.



(a) Reputation value of incorrect nodes



(b) Correctness with our reputation based failure detector

Fig. 8: Accuracy of RepFD with respect to fail silent failures

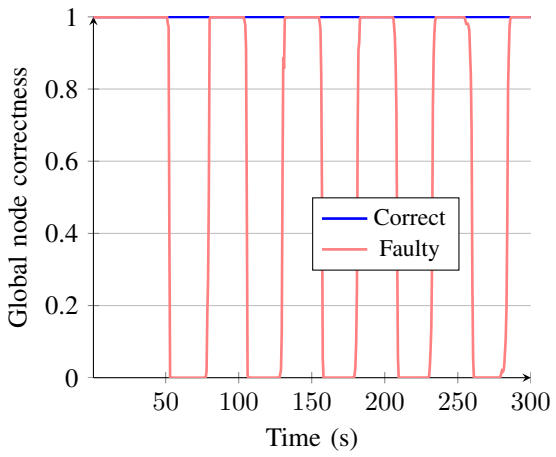


Fig. 9: Accuracy of BertierFD in our crash/recovery scenario

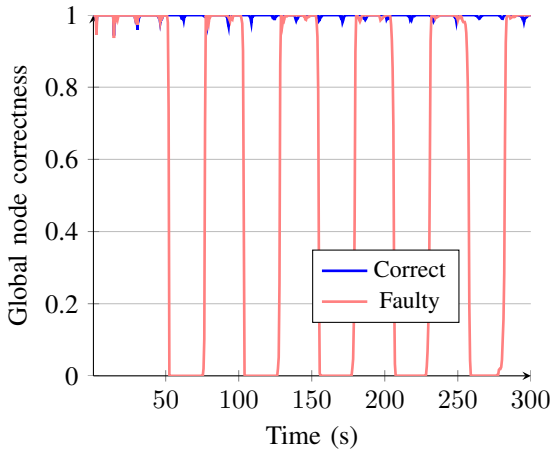
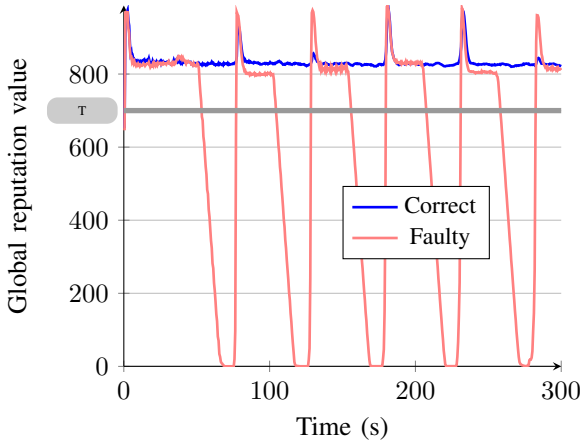
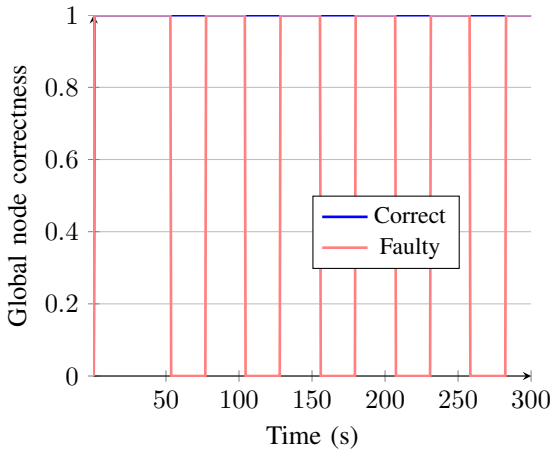


Fig. 10: Accuracy of SwimFD in our crash/recovery scenario



(a) Reputation values in our crash/recovery scenario



(b) Correctness deduced from the reputation assessment

Fig. 11: Accuracy of RepFD in our crash/recovery scenario

F. *Overnet experiment: measuring up to a realistic trace*

In order to test the failure detectors in an environment that reproduces realistic failures, we took the *Overnet* trace from the *Failure Trace Archive* [15]. *Overnet* is representative of the large and dynamic environments we target: it constantly sustains a high number of omissions from all nodes.

During each experiment, every node reads a separate node trace selected at random, and then connects to or disconnects from the system according to the trace. This creates an environment where incorrect nodes omit heartbeat emissions. The average time between two disconnections is 4 seconds; incorrect nodes generate an average of 1 failure every 10 heartbeats. Our testbed produced more than 280,000 node reconnections throughout this series of experiments.

We start by studying the detection time of each detector in this scenario. The presentation of our results differs from previous Subsections because average values would mask the important detection time variations incurred by the detectors. Figure 12 uses boxes to represent the second and third quartile; the crossing line inside the boxes represents the median; and finally the whiskers represent the 90th and 10th percentile. All three detectors produce very similar behaviors in terms of detection time: this confirms the results we obtained for SwimFD and RepFD in the crash/recovery scenario of Subsection IV-E. The high frequency of the reconnections explains that the detection times of BertierFD remain close to those of SwimFD and RepFD. As soon as BertierFD starts suspecting a faulty node, the latter resumes its communications and BertierFD cancels its RTT reevaluation. While this helps BertierFD maintain reasonable detection times, it seriously impacts its accuracy.

G. *Query accuracy probability*

Query accuracy probability (P_A) [11] reflects the probability that a failure detector's output is correct at any random time. We computed the P_A associated with each detector in every experiment. Table 13 compiles the resulting values.

In a failure free context, all detectors exhibit a P_A above 0.9.Yet in this experiment as in all others, the P_A of SwimFD is notably lower than those of BertierFD and RepFD. SwimFD frequently suspects nodes because of the jitter induced by its suspicion list. It spends roughly 10% of its time suspecting

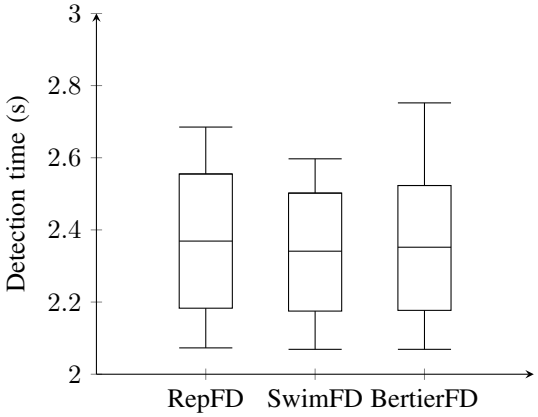


Fig. 12: Overnet - Comparison of the detection times

Experiment	BertierFD	SwimFD	RepFD
No crash	0.999	0.904	0.998
Permanent crashes	0.999	0.886	0.997
Recovery	0.965	0.824	0.997
Frequent omissions	0.854	0.812	0.951

Fig. 13: Query accuracy probability of the studied detectors

nodes due to longer response times and pinging them to infirm the detection.

In the permanent crashes experiment, BertierFD exhibits a P_A that is consistent with its original crash detection results [4]. The accuracy of RepFD also remains constant in this experiment. SwimFD has to produce more messages when a crash occurs. This slightly impacts the whole system, lowering the detector's P_A down from 0.904 to 0.886: a 2% degradation.

In the crash/recovery experiment, both BertierFD and SwimFD incur a noticeable decrease of their P_A: a 3.5% and a 9.7% degradation respectively. As the delay of one detection is reproduced multiple times in the experiment, the accuracy of both state of the art detectors decreases. Note that the P_A for BertierFD remains above 0.96. Our detector actually produces the best P_A value in this experiment. The responsiveness of the PID computation shines when it comes to addressing node recoveries repeatedly over time.

The FTA trace generating frequent omissions impacts strongly on both SwimFD and BertierFD, whereas it bears little impact on our detector. Their respective P_A results verify this observation: whereas our detector keeps a steady P_A value, strong dynamicity impedes the accuracy of BertierFD ($P_A = 0.854$). The accuracy of SwimFD remains the lowest but switching from infrequent crash/recoveries to frequent omissions degrades the accuracy of BertierFD the most: 11% degradation, whereas the accuracy of both SwimFD and RepFD only incurs a 1% degradation. Bertier's RTT estimation does not converge fast enough, hence the important degradation of the accuracy of BertierFD during this experiment. Conversely, since SwimFD does not rely on past detections, the level of dynamicity of the system impacts less on the degradation of its accuracy.

H. Bandwidth usage

Experiment	BertierFD	SwimFD	RepFD
No crash	840.0	100.8	144.1
Permanent crashes	840.0	102.5	144.9
Recovery	840.0	109.2	145.7
Frequent omissions	840.0	336.2	252.0

Fig. 14: Bandwidth consumption of the studied detectors in bytes/s per node

Another important metric for comparing failure detectors is their network cost. Subsection III-C assesses their respective message complexities, but we chose to measure the overhead generated by each detector in our experiments too. To assess this overhead, we measure bandwidth consumption by logging the number of messages sent and by assuming a default size of ping/heartbeat of 84 bytes: 20 bytes for IP headers, 8 bytes for the ICMP header, followed by the default 56 bytes of extra data as specified for Unix ping requests. In the case of RepFD, the size of heartbeats also includes updates of reputation values. The table of Figure 14 compares the network costs of the three studied detectors: it shows the average bandwidth consumption per node and per second in every scenario.

All the values in this table remain low by the standards of current broadband connection networks. However, it is important to remember that we target large and dynamic systems where network overloads are common. Moreover, some of the failures we aim to detect can be induced by such overloads. In this context, a lightweight protocol is paramount for failure detection.

As expected, the all to all heartbeat protocol of BertierFD is the most costly. It generates a constant overhead of nearly 1 kilobyte per second per node, regardless of the scenario.

Under favorable conditions, for instance a low frequency of failures, the optimistic approach of SwimFD is very cost-efficient. On the contrary, in a context of high omissions the suspicion mechanism of SwimFD triggers the degraded mode often. This generates a lot of network overhead, and the costs would rise even higher in a larger network.

RepFD also proves very cost-efficient. Its overhead is slightly higher than that of SwimFD when the frequency of failures is low. But RepFD reacts well to nodes with very erratic heartbeat emissions: it consumes 25% less bandwidth than SwimFD in the Overnet experiment. Two factors contribute to this behavior: the low average degree of nodes imposed by our reputation system, and the fact that variations of the reputation values are piggybacked on heartbeats. An added advantage of this design is that it scales well.

V. Related work

Since the introduction of the notion of unreliable failure detectors by Chandra and Toueg in 1996 [2] there have been many research efforts in this area.

A first class of detectors, such as Swim [6], is based on probing: nodes periodically probe their neighbors by sending ping messages. A second class relies on heartbeats: Bertier [4] and Chen [11] belong to this second class. Several efforts

have been made towards scaling up failure detectors implementations. In [16], a hierarchical topology is used to reduced message complexity. Larrea *et al.* also aim to diminish the amount of exchanged information in order to scale up. To do so, they propose to use a logical ring to structure message exchanges [17]. Finally, Swim [6] scales by using a probabilistic approach: nodes randomly choose a subset of neighbors to probe.

In a recent work, Leners *et al.* propose Falcon [1]. It focuses on the fault detection speed and on the reliability of failure detectors: a process detected as faulty is never up (*i.e.,* always effectively faulty). Falcon may kill components to ensure this property. It is mainly designed to be used among processes within a same local area network (*e.g.,* a single data center).

All the failure detectors presented above classify processes either as correct or as faulty. Accrual failure detectors [3] associate a value with each process, which represents the risk that the process is indeed faulty.

Our failure detector, based on a reputation mechanism, can rely on a more global (yet still fuzzy) view of the system than the state-of-the-art detectors. In this work we use our own reputation system presented in Section II-A to detect faults among nodes. However, other reputation systems such as [10] or systems with more advanced concepts like the ones described in [7], [18] could also act as failure detectors as long as the system is well tuned to detect faults. For short, the aim of this work is not to propose a novel reputation system, but to demonstrate that a reputation system can be used to build efficient failure detectors for large and dynamic networks.

VI. Conclusion

In this paper we present an approach for implementing failure detectors that target large scale networks where dynamic reconfigurations are frequent. We show that a failure detector built on top of a reputation system achieves excellent accuracy and completeness while tolerating high levels of omissions. The exchange of subjective assessments among nodes leads to an efficient cooperation towards an accurate and up-to-date view of failures. Overall, a reputation-based detector combines reasonable detection times with good accuracy when the system runs well, and slows down the local detection just enough to prevent false positives when the system incurs frequent omissions.

We are currently running a large experiment over PlanetLab: we aim to exploit the unreliableness of the nodes to verify the robustness of our approach and its ease of deployment. We also plan to check how our approach withstands network partitions. Another related and forthcoming study is the cooperative exploration of the system to identify multi-nodal points of failure as they occur.

References

[1] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the falcon spy network," in *Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011, pp. 279–294.

[2] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[3] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The phi; accrual failure detector," in *Symposium on Reliable Distributed Systems*, 2004, pp. 66–78.

[4] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *International Conference on Dependable Systems and Networks*, 2002, pp. 354–363.

[5] A. Tomsic, P. Sens, J. Garcia, L. Arantes, and J. Sopena, "2w-fd: A failure detector algorithm with qos," in *International Parallel and Distributed Systems*, 2015.

[6] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 303–312.

[7] A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision," *Decision Support Systems*, vol. 43, no. 2, pp. 618–644, 2007.

[8] A. Rahbar and O. Yang, "PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing," *Parallel and Distributed Systems*, vol. 18, no. 4, pp. 460 –473, 2007.

[9] M. Véron, O. Marin, S. Monnet, and Z. Guessoum, "Towards a scalable refereeing system for online gaming," *Multimedia Systems*, pp. 1–15, 2014.

[10] M. Srivatsa, L. Xiong, and L. Liu, "TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks," in *14th international conference on World Wide Web*, New York, NY, USA, 2005, pp. 422–431.

[11] W. Chen, S. Toueg, and M. Aguilera, "On the quality of service of failure detectors," *Computers*, vol. 51, no. 5, pp. 561–580, 2002.

[12] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 314–329, 1988.

[13] M. Véron, O. Marin, and S. Monnet, "Matchmaking in multi-player on-line games: Studying user traces to improve the user experience," in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, New York, NY, USA, 2013, pp. 7:7–7:12.

[14] A. Montresor and M. Jelasity, "PeerSim: A Scalable P2P Simulator," in *peer-to-peer*, Seattle, WA, 2009, pp. 99–100.

[15] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," in *International Conference on Clusters, Cloud and Grid Computing*, Washington, DC, USA, 2010, pp. 398–407.

[16] M. Bertier, O. Marin, and P. Sens, "Performance analysis of a hierarchical failure detector," in *International Conference on Dependable Systems and Networks*, San Francisco, CA, 2003, pp. 635–644.

[17] M. Larrea, S. Arévalo, and A. Fernández, "Efficient algorithms to implement unreliable failure detectors in partially synchronous systems." in *Distributed Computing*, vol. 1693, Bratislava, Slavak Republic, 1999, pp. 34–48.

[18] A. Ban and N. Linial, "The dynamics of reputation systems," in *13th Conference on Theoretical Aspects of Rationality and Knowledge*, New York, NY, USA, 2011, pp. 91–100.