Context-Aware Fault Tolerance in Migratory Services

Oriana Riva
Department of Computer Science
ETH Zürich
8092 Zürich, Switzerland
oriva@inf.ethz.ch

Josiane Nzouonta Cristian Borcea
Department of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102, USA
{jn62,borcea}@cs.njit.edu

ABSTRACT

Mobile ad hoc networks can be leveraged to provide ubiquitous services capable of acquiring, processing, and sharing real-time information from the physical world. Unlike Internet services, these services have to survive frequent and unpredictable faults such as disconnections, crashes, or users turning off their devices. This paper describes a context-aware fault tolerance mechanism for our migratory services model. In this model, a per-client service instance transparently migrates to different nodes in the network to provide a continuous and semantically-correct interaction with its client. The proposed fault tolerance mechanism extends the primary-backup approach with a context-aware checkpointing process. The backup node is dynamically selected based on its distance from the client and service, the similarity of its mobility pattern with those of the client and service, the frequency of the checkpointing process, and the size of the checkpointing state.

We demonstrate the feasibility of our approach through a prototype implementation tested in a small scale ad hoc network of smart phones. Additionally, we simulate our mechanism in a realistic urban environment with 300 pedestrians, cyclists, and cars. Compared to approaches where the backup node is a neighbor of the service node or the client node itself, our mechanism performs as much as 80% better than the former for recovery ratio, and three times better than the latter for network overhead, while achieving better or similar recovery latency.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems; C.4 [Performance of Systems]: Fault tolerance; D.4.5 [Operating Systems]: Reliability

General Terms: Design, Experimentation, Performance, Reliability

Keywords: Context-aware Fault Tolerance, Migratory Services, Mobile Ad Hoc Networks

MobiQuitous 2008, July 21 - 25, 2008, Dublin, Ireland.

1. INTRODUCTION

Mobile ad hoc networks can enable a new class of ubiquitous services that go beyond simple file transfer applications. These services exploit the temporary and unstable network support of ad hoc networks to provide real-time information acquired from nodes located in the immediate proximity of geographical regions, entities, or activities of interest. Cars on the road can create ad hoc networks to share traffic information, smart phones can collaborate to provide recommendations on city happenings, and intelligent mobile video cameras can cooperate to transmit images from the proximity of a disaster area. However, deploying services in ad hoc networks is extremely challenging due to the dynamism, in particular mobility, and heterogeneity of the interacting entities.

Under these conditions, we believe that service reliability represents the "make it or break it" factor in turning ad hoc networks into distributed service providers. In traditional distributed computing systems, services work correctly for long periods of time, and recovery mechanisms are triggered only in exceptional situations. In ad hoc networks, communication, software, and hardware faults occur frequently and can ultimately render service provisioning unfeasible, especially in the case of long-running stateful operations. Despite node mobility and limited resources, client applications demand stable interactions with services. While numerous approaches [1,4,6] have been proposed to accomplish service reliability in distributed and mobile Internet-based systems, to the best of our knowledge, similar mechanisms have not yet appeared in mobile ad hoc networks. The only fault tolerance issues studied so far in such networks are at the network layer [5, 14].

This paper tackles the service reliability issue in ad hoc networks by proposing a context-aware fault tolerance mechanism for our *migratory services* [22] model. This model supports continuous and stateful client-service interactions in highly volatile ad hoc networks. Unlike a regular service that executes always on the same node, a per-client migratory service instance migrates to different nodes in the network to effectively process its client request. The service migration is triggered by changes of the execution context of the nodes in the network and occurs transparently to the client application. In this way, migratory services naturally provide fault tolerance to mobility and "predictable failures" of nodes. Each time a node becomes unsuitable for hosting a service (e.g., the node has moved away from a region of interest or is running out of battery), the service can autonomously migrate to another node where it can compute

semantically-correct results. However, migratory services do not provide fault tolerance to unpredictable failures occurring when the hosting node crashes, is abruptly turned off, or loses connectivity with the network.

To overcome these problems, we propose a context-aware fault tolerance mechanism for migratory services. Our mechanism extends the primary-backup approach [3], in which one primary service interacts with the clients and one replica is stored on a secondary node. We make the checkpointing process context aware. The secondary node is selected based on its distance from the client and service. Additionally, we take into account its mobility pattern (i.e., current and previous locations and speeds) as well as the checkpointing frequency and state size. Overall, this context-aware, adaptive approach allows the system to make dynamic decisions for constantly trading off between the reliability performance and the induced overhead.

We demonstrate the viability of our approach by implementing a prototype in Java and testing it in a small scale WiFi-based ad hoc network of smart phones. In addition, to investigate the scalability of our mechanism in large scale networks, we simulate it in a realistic urban scenario, where pedestrians, bikers, and cars move across the city roads with different speeds. The evaluation compares the proposed adaptive strategy to two other checkpointing strategies, where the backup node is a random neighbor of the primary or is the client node itself. The results show that our approach reduces the network overhead by two-three times compared to the other approaches. Additionally, they demonstrate that 90-100% of the service interactions, under our approach, successfully recover upon failure even with an increasing number of clients.

The rest of the paper is organized as follows. Section 2 introduces the migratory services model. Section 3 describes the context-aware fault tolerance mechanism for migratory services. Section 4 gives insights on the prototype implementation and its evaluation. The simulation results and analysis are presented in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

2. MIGRATORY SERVICES

Migratory services [22] enable a new model of client-service interaction capable of adapting to the dynamic execution context of mobile ad hoc networks. In response to context changes, a migratory service migrates to different nodes in the network in order to effectively accomplish its task. The service executes on a certain node as long as it is able to provide semantically-correct results; when this is not possible anymore, it migrates through the network until it finds a new node where the execution can be resumed. For instance, to track a suspicious entity, a migratory service can move from one wireless intelligent camera to another as the entity moves across the region, thus constantly providing the end user with images of the subject of interest. While migrating, the service maintains its execution state and a continuous interaction with the client.

The migratory services model incorporates three main mechanisms. The first monitors the dynamism of interacting entities (client or service) by assessing *context* parameters characterizing their state of execution and resource availability. The second specifies, through *context rules*, how the service execution is influenced and should be modified

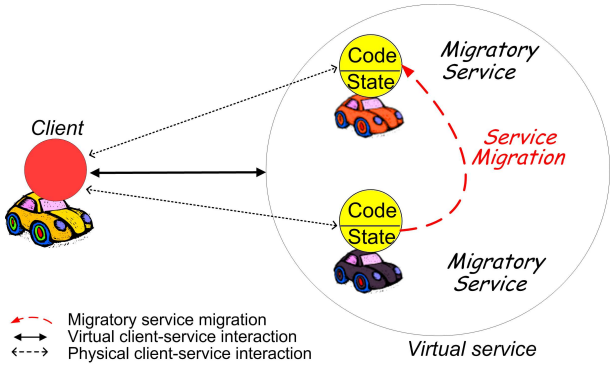


Figure 1. Migratory services model

based on variations of those context parameters. The third makes the service capable of migrating from node to node and of resuming its execution once migrated. We call it *context-aware service migration* since it is triggered by context changes occurring on the service as well as on the client side.

As Figure 1 shows, the service migration occurs transparently to the client, and except for a certain delay, no service interruption is perceived by the client. For instance, the service in the figure may be used by drivers on the highway to predict upcoming traffic jams. This service is required to execute in a region located at a constant distance ahead of the client's car. Due to the mismatch between the speed and direction of the client's car and those of the car executing the service, the region of interest moves and changes dynamically. Consequently, the service may find itself running outside such a region. In response to this context change, it must migrate to another node located inside the region, while carrying its state in the form of short term history of the number and average speed of its neighboring cars.

The figure also shows how a migratory service constantly presents a single virtual end point to the client despite being physically located on different nodes over time. Hence, a continuous client-service interaction is provided. This service model aims to support long-running queries, typical for real-time monitoring scenarios, and can be characterized as "one request, multiple responses".

To facilitate service code distribution in the network, nodes provide *metaservices* for each type of service they own. Clients discover and contact metaservices that process their requests and instantiate a migratory service for each client request. A metaservice just spawns migratable instances of itself, but it does not migrate or send responses to clients. There is a *one-to-one mapping* between a migratory service and a client application. Upon the creation of a migratory service, the client application ceases communication with the metaservice and continues to interact solely with its associated migratory service.

3. CONTEXT-AWARE FAULT TOLERANCE

While the context-aware nature of migratory services improves significantly the client-perceived service quality, service failures are still possible and can thus compromise the

service reliability. A *service failure* can occur due to transient or permanent failures of its components and communication links. Nodes hosting services can crash due to malfunctioning, resource exhaustion, hardware and software errors, or can be deliberately switched off (e.g., to save battery or to reboot). Additionally, communication failures can occur due to mobility.

From a client's perspective, several types of service failures can occur [6]. An *omission failure* occurs when the service omits to respond to a request. A *timing* or *performance failure* occurs when the service responds outside the specified time interval. A *response failure* occurs when the response is semantically-incorrect. In reaction to omission and timing failures, the client sends its request again. If after a certain number of attempts the service omits to produce responses, the service is said to suffer a *crash failure*.

The basic migratory services model naturally handles response failures. A migratory service constantly verifies the correctness of a computed result before delivering it to the client application. If the client node receives a wrong result (e.g., a response computed outside the region of interest for the traffic jam predictor service), this is deleted and a message is sent to the migratory service to update the request parameters. Hereafter, we will focus on providing fault tolerance to permanent crash failures. In the following, we present the two basic components of our context-aware fault tolerance mechanism for migratory services: the backup and the recovery.

3.1 Service Backup

To attain protection against the failures mentioned above and to cope with the volatility of mobile ad hoc networks, we extend the primary-backup approach [3]. A running migratory service, called *primary service*, relies on the existence of one backup service, called *secondary service*. The primary service creates such a backup service, places it on a secondary node, and periodically *checkpoints* the state of the service on the secondary. The secondary service is inactive as long as the client interacts with the primary. If the primary fails, a *failover* process occurs, and the secondary service takes over the service execution. This fault tolerant extension of the service is completely transparent to the client application, which is still provided with the illusion of a single service node.

Fault tolerance comes, however, with the cost of periodic checkpointing. This cost is mostly a communication cost and is proportional to the physical distance between the primary and secondary node. Therefore, in selecting the secondary, we need to consider a trade-off between the desire to minimize the cost of checkpointing and the reliability level that can be provided. On the one hand, the secondary node should be close to the primary in order to minimize the checkpointing overhead. On the other hand, it should be close to the client in order to improve the chances of a failover and minimize the failover latency. In the extreme case, the secondary service may reside on the client node itself.

In practice, a secondary node close to the client is preferred for highly critical applications that demand high reliability and fast recovery. However, when the service state is relatively large or the checkpointing frequency is high, a secondary node close to the primary is preferred (e.g., in a tracking application, a service state consisting of large image files should be saved on the closest nodes). Based on these considerations, the ideal distance d between the primary and the secondary is computed as follows

$$d_f = \begin{cases} d_{PC} & \text{if } f \leq f_{min} \\ 0 & \text{if } f \geq f_{max} \\ d_{PC} \cdot \frac{f - f_{max}}{f_{min} - f_{max}} & \text{otherwise} \end{cases}$$

$$d_s = \begin{cases} d_{PC} & \text{if } s \leq s_{min} \\ 0 & \text{if } s \geq s_{max} \\ d_{PC} \cdot \frac{s - s_{max}}{s_{min} - s_{max}} & \text{otherwise} \end{cases}$$

$$d = \alpha \cdot d_f + (1 - \alpha) \cdot d_s \qquad \alpha = 1/2$$

The distance d is computed by a weighted combination of d_f and d_s. d_f is the ideal distance between the primary and the secondary based on the checkpointing frequency f. d_s is the ideal distance based on the service state size s. The minimum and maximum threshold values for frequency and state size are set statically by the primary node taking into account the frequency requirements and state characteristics of typical migratory services. For instance, the minimum state size could be associated with an average car speed service, and the maximum could be associated with an entity tracking service that uses multiple images and performs image recognition.

As f varies from f_{min} to f_{max}, d_f varies linearly from d_{PC} to 0, where d_{PC} is the distance between the primary and the client. Below the lower threshold f_{min}, d_f equals d_{PC}, meaning that the checkpointing frequency is so low that the client can act as secondary. Above the upper threshold f_{max}, d_f is 0, meaning that the checkpointing frequency is so high that the secondary should be located on a neighbor of the primary node. Similar considerations hold in computing d_s. The parameter α, set to 1/2 by default, can be tuned to change the impact of the state size and checkpointing frequency on the computation of the secondary's location.

However, selecting a secondary at the ideal distance from the primary does not necessarily mean to have found an adequate secondary node. We improve the selection process by considering the available resources and mobility traces of all potential secondary nodes. We call this type of information *context* of the node. In principle, the primary could monitor all context parameters of all nodes located in the routing path to the client within the ideal distance d and then select the most appropriate secondary among them. However, a monitoring process of this type would turn out to be too expensive and not scalable. Instead, we adopt an on-demand solution in which the evaluation is distributed over the network.

First, as previously explained, the primary node computes the ideal position of the secondary node on the routing path between itself and the client. Then, it sends a *sec-discovery* message in the network. This message contains the current location and mobility traces of the primary node, the location of the client, and a list of N requirements that the secondary node should satisfy. These requirements are specified in a vector of conditions $<c_1, ..., c_N>$, where each condition expresses lower bounds on needed resources, such as CPU, memory, and communication capabilities. The *sec-discovery* message is broadcast in a geographical region of range r centered on the ideal position of the secondary. r is defined in

such a way to generate the maximum discovery region that does not expand beyond the client's and primary's positions. To avoid regions that are too small and potentially do not contain any node, r cannot be lower than r_{min} (set to twice the wireless transmission range t_r).

$$r = max\{r_{min}, min\{d, (d_{PC} - d)\}\} \qquad r_{min} = 2 * t_r$$

Upon receiving a *sec-discovery* message, each node i checks whether its characteristics match the requirements specified in the message and computes a matching score s_i

$$s_i = \beta \cdot match(mob_i, mob_P) + \frac{1-\beta}{N} \sum_{j=1}^{N} c_j(p_{i,j})$$

Since we consider mobile nodes, the reliability and quality of the communication can highly improve if the primary and secondary nodes constantly move in the same direction and at similar speed. The *match* function matches the mobility traces of the candidate secondary node i and the primary node, namely mob_i and mob_P. Each trace contains the location coordinates $< latitude, longitude >$ of the node over the last N minutes, sampled every x seconds. GPS receivers, for instance, can provide these traces and navigator systems can even record the most frequently taken routes (e.g., home-to-office, office-to-home). *match* computes the average distance between the positions of the two nodes over the last M minutes (e.g., 15 minutes); if the result is below a certain threshold, it returns 1, otherwise 0.

The second part of the score computation considers the device profile (e.g., CPU, memory, and communication capabilities). Secondary nodes with larger capabilities are desirable because they are likely to be operative for longer and be effective in promptly resuming the service execution when a fault occurs. Each condition c_j is evaluated against the corresponding profile parameter $p_{i,j}$ of the node i; if positively verified, 1 is returned, otherwise 0. The matching score is a weighted combination of the two matching outputs. The parameter β, set to 1/2 by default, can be set depending on the execution environment such as device types and mobility. For example, in scenarios where nodes are computers embedded in cars, matching mobility traces is more relevant as nodes move fast and they are not resource-constrained. Thus β can be close to 1. As opposite, in scenarios where nodes are smart phones carried by walking people, the resource-based matching is more relevant and β can be close to 0. Nodes with matching scores higher than a certain fixed threshold reply to the primary with their score values and profile information. The primary can then select its secondary node.

Once a secondary node is selected, *state-update* messages are exchanged between the primary and secondary nodes such that if the primary fails, the service state is available on the secondary. The service state consists of service data and execution control state. A checkpoint of the service state occurs every R computed responses. This choice ensures that every checkpointed state contains information "relevant enough" to be saved. If R is high, the checkpointing overhead is reduced, but with the risk of having an inconsistent state on the secondary. If R is low, the overhead increases, but the reliability improves. Every *state-update* message must be acknowledged by the secondary node.

State-update acknowledgments can be used to carry control information about the secondary's current context, such as distance in number of hops between primary and secondary, remaining battery power, status of network connectivity, etc. This control information may be used to trigger a new secondary selection process or a migration of the secondary to a more suitable node when necessary. Likewise, the checkpointing frequency may be adapted accordingly, thus better tolerating transient network disconnections and optimizing resource utilization.

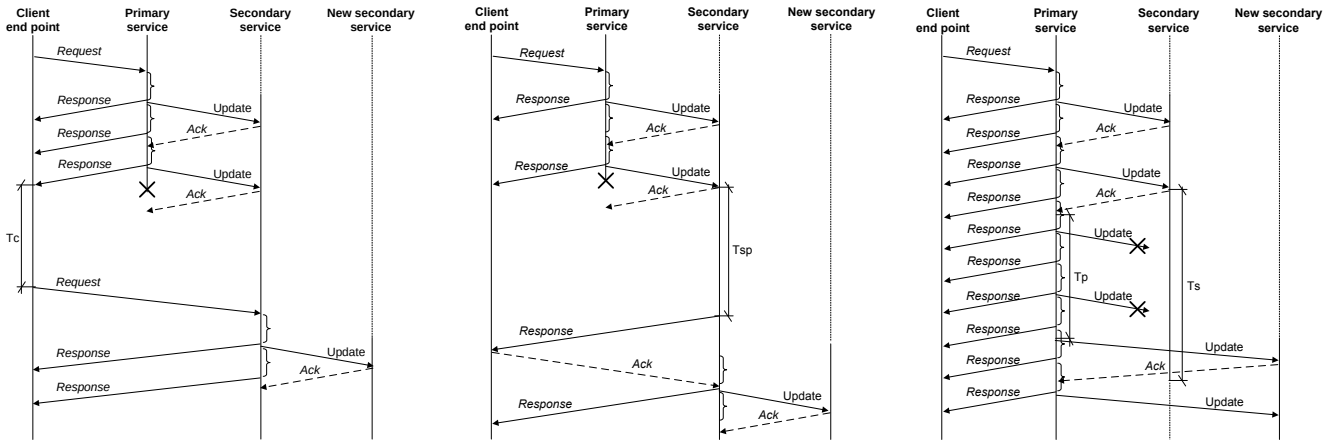
3.2 Service Recovery

Since it is hard to maintain a supervising entity in highly volatile ad hoc networks, node failures are detected by mutual monitoring, and the recovery is achieved collaboratively. The recovery process can be pull-based or push-based, depending on which node detects the failure and initiates the recovery. In *pull recovery*, the client node assumes that the primary has crashed when it stops receiving answers. After a certain number of failed attempts to reconnect to the primary, the client contacts the secondary which will undertake the recovery. Pull recovery is generally employed by monitoring services that follow a "one request, multiple responses" model where responses are periodically sent to the client. *Push recovery*, on the other hand, is specific to situations where the client cannot tell whether a fault has occurred, such as in the case of event-based services (i.e., the client cannot tell whether it should or should not have received a response). In this case, the secondary detects the failure, activates the recovery, and resumes the interaction with the client. The secondary assumes a primary fault has occurred when it stops receiving *state-update* messages.

Failures of the primary service and disconnections between the primary and secondary are detected using timeouts. To adapt to changing network conditions, timeouts are constantly updated, but always in a range between a minimum and a maximum value set at the beginning of the interaction. Ideally, only persistent disconnections or node crashes should trigger the replacement of the primary service with the backup service. Timeouts should be set and dynamically adjusted with the aim to quickly detect permanent failures and tolerate transient ones. In our model, we define four timeouts:

- T_c is the client timeout for pull recovery. The client concludes that the primary has crashed if it did not receive any response during this period.
- T_{sp} is the secondary timeout for push recovery. The secondary concludes that the primary has crashed if it did not receive any *state-update* message during this period.
- T_p is the primary timeout for both types of recovery. The primary concludes that the secondary has crashed if it did not receive any *state-update acknowledgment* during this period.
- T_s is the secondary state removal timeout. The secondary considers that the primary has selected a new secondary if it does not receive any *state-update* message from the primary or a request from the client to take over during this period.

Figure 2 illustrates three main types of failure scenarios and how the timeouts are used to cope with them. The



(a) Primary failure and pull recovery (b) Primary failure and push recovery (c) Secondary failure

Figure 2. Failure scenarios

first case, Figure 2(a), shows a pull recovery. In this scenario, the primary service periodically delivers responses to its client. At some point, the primary crashes and disconnects from both the client and secondary. When T_c expires, the client concludes that the primary has crashed and resubmits its request to the secondary. The secondary assumes the role of primary, resumes the interaction, and selects a new secondary node. T_c is proportional to the estimated RTT (round trip time) between the client and primary, and RTT is constantly updated based on the jitter of the observed interresponse times. Furthermore, to guarantee that the client times out before the secondary removes the service state, T_c must be less than T_s.

Another possibility, not shown in the figure, is that the primary service reappears after some time by sending a new response to the client and a new update to the secondary. In this case, both the client and the secondary reply to the primary with a *cancel* message, and the client increases its T_c by 1 RTT to better cope with current communication delays.

The second scenario, shown in Figure 2(b), shows a push recovery. This example considers an event-based service interaction. As the primary does not send periodic responses to the client, the client cannot directly detect its failure. However, the secondary can easily realize when the primary has failed (i.e., it does not receive *state-update* messages). When T_{sp} expires, the secondary contacts the client; if the client also believes the primary has crashed, it replies by asking the secondary to take over the execution. To guarantee that recovery occurs before the secondary removes the service state, T_{sp} must be less than T_s. If the primary appears again after some time, both the client and secondary will reply with a *cancel* message and increase their timeouts by 1 RTT. If the client believes the primary is still active (i.e., the primary has probably disconnected only from the secondary), no reply is sent to the secondary. Consequently, T_s will expire and the secondary will remove the service state. In this case, the primary node guarantees that a new secondary has been selected by the time the old secondary deletes its state as demonstrated in the next example.

Figure 2(c) shows a scenario in which the secondary fails. When T_p expires, the primary selects a new secondary node.

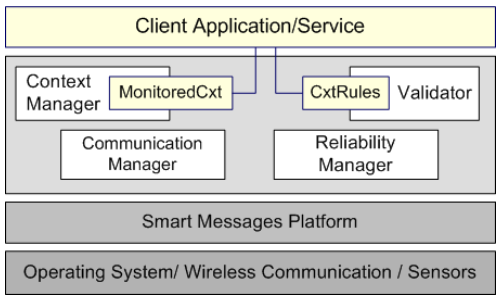


Figure 3. Migratory services framework

Like in the previous case, the timeout T_p is computed and updated based on the observed primary-secondary RTT. In the case of a temporary disconnection of the secondary, it may happen that the second lost update shown in the figure reaches the secondary node and triggers an acknowledgment. If this acknowledgment arrives after T_p has expired and a new secondary has already been selected, the primary ignores it, thus forcing the old secondary to timeout. In addition, the primary increases its T_p by 1 RTT as it is likely that other transient disconnections will happen soon. T_p will be decreased again if no other disconnections will occur for a while.

4. PROTOTYPE SYSTEM

To support the migratory service model, we implemented a framework that runs on each node willing to cooperate in the ad hoc network. As illustrated in Figure 3, at the lower layer, the Smart Messages (SM) [11] computing platform provides support for execution migration, naming, routing, and security. On top of the SM layer, we built a layer providing support for context provisioning and monitoring, context rules creation and validation, client-service communication, and service reliability. This layer exports a simple message passing API to clients and services. This framework was implemented using the mobile and embedded Java platform (Java ME) Connected Device Configuration (CDC) 1.0. A portable SM [21] version that runs on top of unmodified Java

virtual machines was used in the implementation. All software development was done using Nokia Series 80 phones.

To understand the implementation details of our prototype, we start this section with an overview of the SM platform. We then describe how the fault tolerance mechanism has been implemented in the migratory services framework and evaluate its performance.

4.1 Smart Messages

An SM is an application similar to a mobile agent whose execution is distributed over a series of nodes using execution migration. The nodes on which SMs execute are named by properties and discovered dynamically using application-controlled routing. To move between two nodes of interest, an SM calls explicitly for execution migration. Each node participating in the SM execution provides a *virtual machine* for execution over heterogeneous platforms and a shared memory addressable by names, namely the *tag space*, for SM communication, synchronization, and interaction with the host.

During execution, an SM can interact with the host or other SMs through the tag space, which is local to each node. The tag space consists of *(name, data)* pairs, called tags, which are created by SMs and used for data exchange. The tag space also provides a simple update-based synchronization mechanism: an SM can "block" on a tag until another SM "writes" on that tag. Special I/O tags are predefined at nodes and used as an interface to the OS and I/O system (e.g., battery lifetime, available memory, sensors). Tags are also used to uniquely name both end points of a communication, as well as to define the destination node of a service migration. SM integrates geographical routing and region-bound content-based routing, similar to GPSR [12] and AODV [20], respectively.

The multi-hop SM migration is implemented using a low-level primitive for one-hop migration, namely *sys_migrate*. This primitive captures the execution context of the SM (data and control state), packs it with the SM code, transfers all to the next hop, and resumes the execution with the following instruction in the code. SMs are Java programs, which can incorporate multiple Java classes, namely *code bricks*, and multiple Java objects, namely *data bricks*. The data bricks are explicitly specified by the programmer when an SM is instantiated and contain the data that must be transferred during migrations. At runtime, SMs can create "child" SMs carrying a subset of their code bricks and data bricks.

4.2 Reliability Modules

Our context-aware fault tolerance mechanism has been integrated in the migratory services framework through the *Reliability Manager*. Figure 4 shows the core software modules composing the Reliability Manager and the interactions that take place between the primary and secondary node during the backup and recovery process. The primary service generates a `SecondaryDiscoverySM` to discover a qualified secondary node. This SM is broadcast to all candidate secondary nodes located in the target geographical region computed by the primary's framework. Instances of this SM verify whether any node in the region meets the specified requirements and migrate back to the primary when an

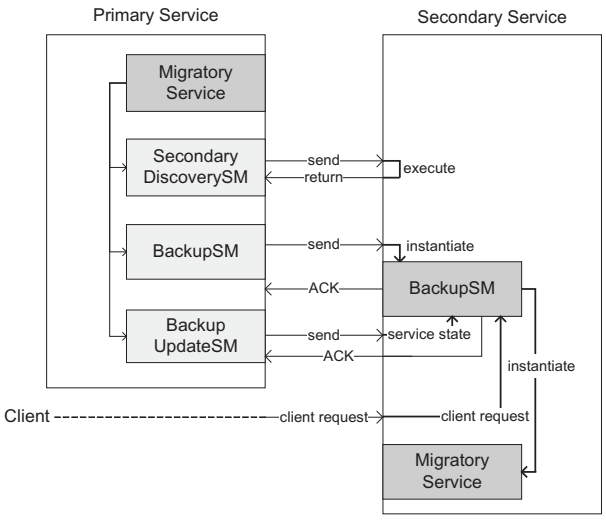


Figure 4. Interaction between the reliability software modules

appropriate secondary is found. Once a secondary node is selected, the primary service generates a `BackupSM` that migrates to it. The `BackupSM` carries the code bricks and data bricks necessary to instantiate and execute the migratory service on such a node.

The `BackupSM` waits for updates or client requests by blocking on `update` and `request` tags. `Update` tags are used to receive state updates from the primary. These updates are encapsulated in `BackupUpdateSMs`. These SMs migrate to the secondary node and unblock the `BackupSM` by writing an `update` tag containing the new service state. The `BackupSM` reads the tag, updates the state, and acknowledges the reception of the message. `Request` tags are used by the client framework to trigger pull recoveries upon failure of the primary. In such a case, a `BackupRequestSM` (not shown in the figure) migrates to the secondary and writes a `request` tag containing the request parameters and the sequence number of the last response received by the primary. The `BackupSM` unblocks and instantiates a new `MigratoryService` that will take over the execution. In push recovery an analogous SM migrates from the secondary to the client.

4.3 Experimental Evaluation

To demonstrate the feasibility of our approach, we tested our prototype in an ad hoc network of three Nokia 9500 smart phones. The phones communicate using IEEE 802.11b in ad hoc mode. They run Symbian OS 7.0s, have an ARM processor at 150 MHz, offer 76 MB of built-in memory, and support JavaME CDC (JSR-36). As our primary goal was to understand if such a fault tolerance mechanism can run with reasonable performance on smart phones, we did not attempt to optimize the code. A detailed performance analysis in large scale networks is given in Section 5.

In the experiments, we employed TJam, a prototype migratory service for traffic jam prediction. This application dynamically predicts if traffic jams are likely to occur in a given region of a highway by using only car-to-car short-range wireless communication. In the test topology, the phone running the TJam client and the one running the

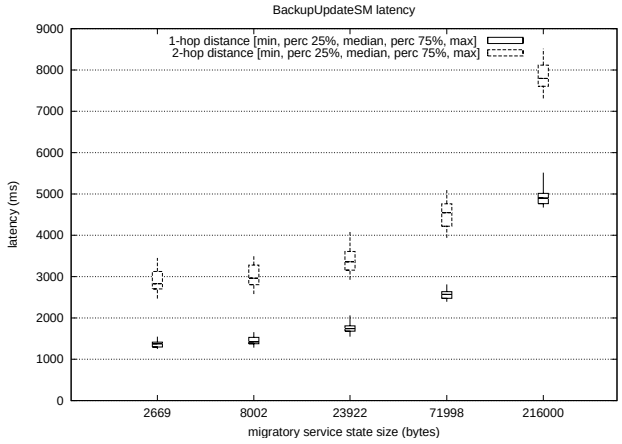


Figure 5. Checkpointing latency for one hop and two hops distance

TJam primary service are at a distance of two hops. The initial service state of TJam accounts for 2669 bytes. This is increased during the tests to assess the performance of migratory services with different state sizes. We ran experiments measuring the memory overhead, checkpointing latency, and recovery latency.

Memory overhead. In these experiments, we measure the heap size differences before and after the objects have been allocated. However, we need to consider that the JVM can decide to increase its current heap size at any time and in particular when running garbage collection. We therefore force garbage collection to happen before the measurements start and ensure that it does not interfere with them[1]. The heap size allocated to run the Java VM and the migratory services framework with TJam running is 172 kB. For a service of 2669 bytes like TJam, the size of the TJam migratory service's client is 68.8 kB, while that of the primary (i.e., the secondary has the same size) is 111.7 kB. Technical specifications say that Nokia 9500 phones offer approximately 20 MB of free RAM to run programs. However, benchmark tests[2] show that the size of the heap is approximately 13 MB. Based on these benchmarks, our experimental migratory service consumes about 0.89% of the available RAM. Therefore, we conclude that even with a relatively small amount of memory, many migratory services can be supported by the system.

Checkpointing latency. This latency includes the time to extract the service state, save it in the `BackupUpdateSM`, migrate the `BackupUpdateSM` to the secondary node, store the new service state in the `BackupSM` running on the framework at the secondary node, and send an acknowledgment to the primary service. We measure the elapsed time from the beginning of the checkpointing process until the reception of the secondary's acknowledgment. We consider two cases: the secondary is at a distance of one or two hops from the primary. In the 1-hop case the secondary runs on the intermediate node between client and primary, while in the 2-hop case it runs on the client node.

[1]We used a modified version of the Sizeof class available from http://www.javaworld.com/javaworld/javatips/jw-javatip130.html

[2]http://www.club-java.com/TastePhone/J2ME/MIDP_Benchmark.jsp

Table 1. Latency of the recovery process

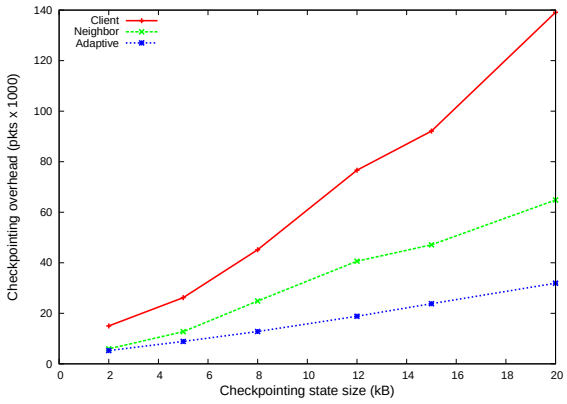
MS State (bytes)	Failover (ms)	1-hopSecDisc (ms)	BackupSM-MigrExAck (ms)
2669	256	1850	7187
8002	256	1850	7234
23922	256	1850	7468
71998	256	1850	8296
216000	256	1850	10704

Figure 5 reports checkpointing latencies for different sizes of the service state. The actual size of the data transfer accounts for data bricks and code bricks of the `BackupUpdateSM`. Together with the service state, data bricks contain also other control information, such as routing, that account for 691 bytes. As these tests were executed without caching the code, code bricks of the `BackupUpdateSM` accounts for 7018 byte.

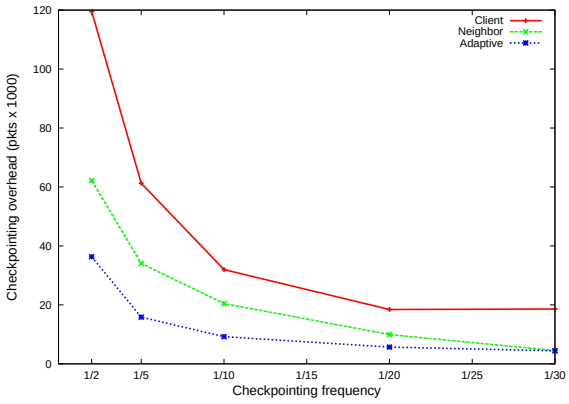
The backup latency increases linearly with the size of the service state. Compared to the case of backup at a distance of one hop, the latency of the backup process at a distance of two hops shows a faster raise as the service state size increases. The break up analysis shows that extracting the service state and generating the `BackupUpdateSM` accounts for approximately 280 ms, while storing the received service state and generating the acknowledgment message accounts for approximately 415 ms. The rest is SM communication overhead, which includes connection establishment, transfer of data and code, and serialization. The SM communication time can be improved if we enable the existing code caching mechanism in SM and use a more advanced serialization mechanism than the default Java serialization based on slow reflection. In all experiments, we observed that Java serialization consumed over a quarter of total time when transferring 1 kB messages over WiFi. Hand-optimized binary formats can improve performance, but they often lack extensibility and are hard to debug. However, recent work on XML encoding [28] shows that it is possible to retain the benefits of a generic format such as XML without sacrificing efficiency.

Recovery latency. To measure this latency, we generate phone failures by turning off the WiFi interface for a period long enough to trigger the timeout on the client node (pull case) or on the secondary node (push case). Table 1 presents the results as function of the migratory service's state size (*MS state*). The latency of the recovery process on the secondary node consists of *(i)* resuming the service execution (*failover*); *(ii)* discovering a new, suitable secondary node based on the requirements of the primary service (*1-hopSecDisc*); and *(iii)* creating and migrating the `BackupSM` that will start executing on the new node by first sending an acknowledgment back to the primary (*BackupSM-MigrExAck*). The size of the `BackupSM`'s data bricks is given by the service state and the additional overhead for routing, discovery, and migration functions that account for a total of 2364 bytes. The `BackupSM`'s code bricks amount to 28757 bytes (these experiments are also run without code caching).

The failover time and the one-hop discovery time are constant as they are independent of the state size. The cost



(a) No failures, frequency=$\frac{1}{5}$



(b) No failures, state size=10 kB

Figure 6. Checkpointing overhead

Table 2. Average users' speed in the simulation study

Users	Percentage	Average speed
Standing	10%	$0m/s$
Walking	20%	$1m/s$
Running	10%	$3.5m/s$
Cycling	10%	$5m/s$
Driving	50%	$18m/s$

of the recovery process is due mainly to the time necessary to migrate data and code of the backup service to the new secondary node. Even though this latency can be high for services consisting of large data bricks and code bricks, it does not directly impact the performance observed by the end user. The delay of service responses sent to the client depends on the failover latency, which we found to be small, and on the timeouts detecting the primary failure.

5. SIMULATION RESULTS

The objective of the simulation study is to analyze the performance of our fault tolerance model in large scale networks. Specifically, it provides quantitative answers to the following questions: What is the network overhead associated with the fault tolerance mechanism? How does our mechanism scale with an increasing number of clients? How long does the recovery process take and how many services do recover?

We use the *NS-2* simulator enhanced with the CMU wireless extensions. The transmission range of each node is 250 m and the propagation model is the TwoRayGround. We consider a 6000 m x 1000 m urban area covering a grid type road layout with 55 bidirectional road segments. A road segment is defined by two consecutive intersections on the same road. We utilize 300 nodes initially distributed uniformly over the roads. The nodes move at different speeds with the aim of representing a realistic urban population consisting of pedestrians, runners, cyclists, and drivers. The speed of each node is modeled as a uniform variable. The percentage of nodes in each category and their average speed values are reported in Table 2.

Among the 300 nodes, 36 nodes are randomly selected to act as *metaservices* (there are three types of metaservices to choose from). As explained in Section 2, metaservices are responsible for receiving client requests and instantiating migratory services. Each request is associated to one migratory service instance. In our simulations, clients ask migratory services to monitor a certain region at a constant distance from its current position (e.g., entity tracking for a static user, traffic jam predictor for a driver). The migratory service must therefore ensure that observations are constantly collected in the region of interest despite node mobility. Each service computes and returns a new response every 3 seconds.

Metaservices and secondary nodes are discovered using geographical forwarding to reach the region of interest and a broadcast inside the region. The rest of communication uses only geographical forwarding. To reduce the overhead associated with traditional geographical forwarding (i.e., frequent "hello" messages to maintain an accurate list of neighbors), we employ our distributed receiver-based next hop election [17]. This solution uses the default IEEE 802.11 RTS/CTS mechanism and a multi-criteria prioritization function, which accounts for non-uniform radio propagation, to allow the self-election of the best neighbor.

Each simulation test runs for 300 seconds and results are averaged over five runs. Each simulation run is produced using a different initial node distribution. We consider the following metrics:

- *Checkpointing overhead* measures the network traffic overhead induced by the fault tolerance mechanism.
- *Recovery ratio* measures the percentage of client-service interactions that are able to recover upon a primary failure.
- *Recovery latency* measures the time elapsed between the moment at which the failure is detected (by the client node in the pull model or by the secondary node

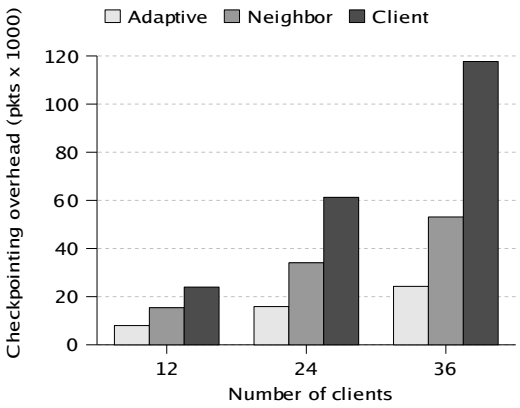


Figure 7. Checkpointing overhead with an increasing number of clients (No failures, frequency=$\frac{1}{5}$, state size=10kB)

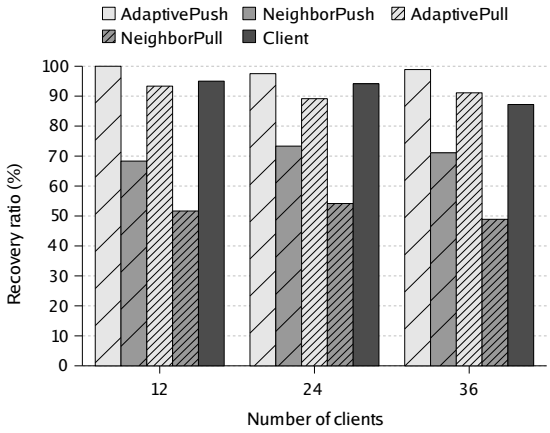


Figure 8. Recovery ratio with an increasing number of clients

in the push model) and the moment at which the normal operational state is reestablished (i.e., the primary service is reestablished and a new secondary node is successfully selected). This metric considers only recovered service interactions.

- *Response ratio* is the number of responses received by the clients in the presence of failures over the number of responses received in the ideal case where no node failure, routing delay, and network partition occur.

The evaluation is carried out by comparing our context-aware fault tolerance mechanism (labeled *Adaptive*), against three other mechanisms, namely checkpointing at the client (*Client*), checkpointing at a random neighbor of the primary service (*Neighbor*), and no checkpointing (*Baseline*). Unless otherwise specified, the parameters for our context-aware fault tolerance mechanism are set as follows: $f_{max} = 1$, $f_{min} = \frac{1}{100}$, $s_{max} = 100100$, $s_{min} = 100$, $\alpha = \beta = \frac{1}{2}$, $T_p = (\frac{1}{f} + 3)\tau$, $T_{sp} = (\frac{1}{f} + 2)\tau$, $T_c = 3\tau$, where τ represents the interresponse generation time and is set to $3s$. Matching of mobility traces is enabled, but we do not study the impact of matching resource capabilities in the secondary selection.

5.1 Checkpointing Overhead

We assess the checkpointing overhead for different checkpointing frequencies and state sizes. The number of clients is fixed to 24. No service failures are induced in these tests as our goal is to quantify the added overhead when fault tolerance is not needed. However, service failures may still occur because of network disconnections or failed service migrations. Figure 6 summarizes the results.

First, we keep the checkpointing frequency constant at $\frac{1}{5}$ (i.e., the checkpoint occurs once every five generated responses) and vary the state size. As the results in Figure 6(a) show, our Adaptive strategy performs best (as much as three times better than the Client strategy and about twice than the Neighbor strategy) and scales better than the others as the state size increases. This occurs because our mechanism always attempts to select backup nodes located at an ideal distance from the primary and client, and it takes into account the checkpointing frequency and state size. Furthermore, the backup node is selected by taking into account similarities between its mobility trace and that of the primary.

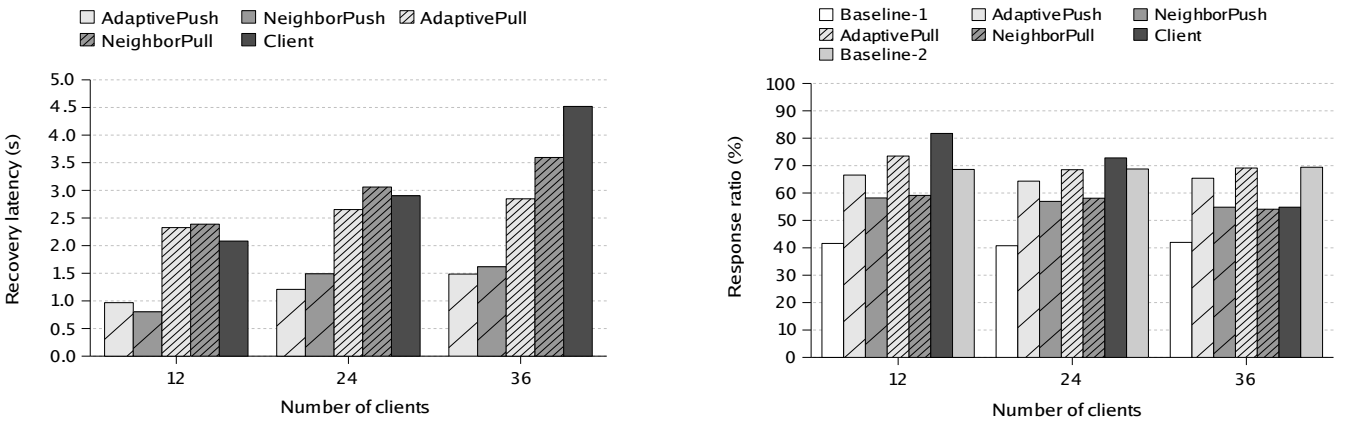
The overhead of the Client strategy increases significantly with the state size because of the longer path between the primary and secondary, which leads to higher rates of packet contentions and retransmissions. Unexpectedly, our strategy outperforms checkpointing at the neighbor as well. This is because the Neighbor strategy selects the secondary without considering the similarities between mobility traces. For instance, the primary service running on a fast car can select a secondary running on a pedestrian node. As nodes are moving, the secondary can easily move far from the primary in a short period of time, thus increasing the chances of disconnection between the secondary and the primary.

Similar considerations explain the results shown in Figure 6(b), in which we keep the state size constant and vary the checkpointing frequency. Our Adaptive strategy is always better than the other two. This is especially visible for high checkpointing frequency (e.g., $f = \frac{1}{2}$, checkpointing is performed once for every two new responses computed).

We also measure the checkpointing overhead as function of the number of clients. Figure 7 demonstrates that our strategy scales better with the number of clients than the other two.

5.2 Recovery Performance

To investigate the recovery performance, we measure the recovery ratio, the recovery latency, and the average response ratio as function of the number of clients. Failures are induced at time t=150 seconds by switching off all primary services. Our mechanism works for both hardware failures (node crashes) and software failures (service crashes). We choose the latter for these simulations to avoid inducing failure effects on routing. Although other failure models in which failures are distributed over time could have been considered, we choose to perform our tests in the worst case scenario, where all failures occur at the same time. Throughout the simulations, the backup services do not fail.



(a) Recovery latency

(b) Response ratio

Figure 9. Recovery latency and response ratio with an increasing number of clients

Furthermore, to make the analysis more realistic, these simulations add background traffic to account for other potential applications in the network (10 random nodes exchange Constant Bit Rate traffic at a rate of 1 pckt/s). The background traffic was not enabled in the overhead tests as we wanted to quantify the overhead induced by the checkpointing process rather than the routing.

We compare the performance of the three mechanisms where Neighbor and Adaptive are present in both the push (*AdaptivePush* and *NeighborPush*) and pull (*AdaptivePull* and *NeighborPull*) versions. In the Client strategy, as the secondary resides on the client node, there is no difference between the push and pull models.

Recovery ratio. As Figure 8 shows, more than 90% of the client-service interactions recover after failure in all tests for the Adaptive and Client strategies. In the Neighbor tests, the recovery rate is between 50% and 75%. This is due to the random selection of the secondary without matching the mobility traces, which leads to longer communication paths in many situations. Consequently, longer paths lead to a higher risk of communication breaks. If the primary does not detect the break in the communication with the secondary in time to select a new secondary, the service interaction will not recover upon the primary failure.

The Client strategy can recover easily because the client and secondary reside on the same node. However, we observe that the Adaptive strategies perform better than the Client in several cases and that the Client's recovery is not 100% as expected. This is due to the fact that a complete recovery requires the backup service to migrate to the monitored region, which can be far away, thus increasing the chances of migration failures. Instead, in the Adaptive strategy the backup node is generally closer to the region of interest, thus reducing the probability of migration failure.

We also observe that the recovery ratio is higher in push scenarios compared to pull scenarios. In the push case, after a primary failure, the secondary times out and directly resumes the interaction with the client. In the pull case, the client detects the failure and contacts the secondary, which will then resume the interaction. This "extra step" of contacting the secondary service in the pull mechanism generally leads to higher recovery latencies. Typically, the distance between the secondary and the region of interest increases over time (due to the speed mismatch in the Neighbor case and due to deviations from the historical mobility trace in the Adaptive case). Therefore, higher recovery latencies lead to longer distances, and consequently, to higher rates of migration failures when the secondary attempts to migrate to the region of interest.

Recovery latency. Figure 9(a) shows that the Adaptive strategy achieves the best recovery latencies. The performance of the Client strategy is affected by its greater distance from the region of interest (i.e., it takes longer to migrate the secondary there). We also observe that as the number of clients increases, its relative performance decreases faster when compared with Adaptive and Neighbor. This is due to wireless contentions and retransmissions, which lead to longer delays in the migration process. The Neighbor strategy achieves performance closer to the Adaptive strategy, but its main problem is the low recovery ratio (i.e., in computing the latency, we considered only the recovered client-service interactions). Finally, the push model achieves lower latency than the pull model due to the "extra step" in the recovery process mentioned above.

Response ratio. Figure 9(b) reports the average response ratio for all mechanisms and for two Baseline cases. This metric quantifies the response losses incurred on the client because of the failures. We choose to measure this metric instead of computing it analytically (from recovery latencies and service interresponse time) to account for packet losses and migration failures. *Baseline-1* represents migratory services without reliability support and with induced failures. *Baseline-2* represents migratory services without reliability support and without induced failures. All numbers are normalized to the number of received responses in the ideal case (i.e., no failures, no network partitions, no

packet losses). Specifically, in the ideal case, the number of expected responses is $\frac{T}{\tau}$, where T is the total simulation time and τ is the service interresponse time. With our settings (T=300 s, τ=3 s), this number is 100.

Our Adaptive strategy achieves the best ratios when the number of clients increases, comparable with the Baseline-2 (the case without induced failures). Furthermore, it achieves as much as 65% improvement compared to Baseline-1. The Client strategy works best for a small number of clients because the reaction time to a failure is reduced due the fact that the secondary resides on the client node. The other methods react slower depending on the timeouts at the primary (T_p) and secondary node (T_{sp}). Note that the recovery latency does not account for how quickly the failure is detected, but for how quickly the recovery occurs once the failure is detected. This explains why in the case of 12 clients AdaptivePush presents a lower response ratio than Client even though its average recovery latency is much lower (Figure 9(a)) and its recovery ratio just slightly higher (Figure 8). On the other hand, the response ratio of the Client strategy quickly decreases with an increasing number of clients due to the increase in contention. The Neighbor strategy performs worst due to its low recovery ratio.

Even though the push model guarantees higher recovery ratio and lower recovery latency, its overall number of received responses is slightly lower than that of the pull model. This result demonstrates the impact on the overall performance of the time to detect a failure. The push model takes longer due to the fact that the secondary's timeout (T_{sp}) employed in push recovery is higher than the client's timeout (T_c) employed in pull recovery.

5.3 Results Summary

These results prove the effectiveness of our approach. We expected the Client strategy to provide the best recovery at the cost of a higher checkpointing overhead. The results validate the overhead assumption (the Adaptive strategy has as much as three times lower overhead), but they also show that the Adaptive strategy achieves slightly better recovery ratios. Additionally, the Adaptive strategy scales better with an increasing number of clients. The Neighbor mechanism was expected to provide a low overhead at the cost of a decrease in the recovery latency. The results demonstrate that the Adaptive strategy achieves not only lower recovery latency, but also lower overhead and higher recovery ratios. Finally, the performance of our mechanism could be improved even further by allowing the secondary to migrate and follow the movement of the primary and the client.

6. RELATED WORK

Reliability in distributed systems can be achieved by avoiding single points of failure, allowing monitoring, reducing the scope of failure to a recoverable action, and especially enabling redundancy. Two well-known types of redundancy are active replication and primary-backup [6]. In active replication, a collection of servers (replicas) maintain the service state, and upon a server failure, the remaining servers can continue. Primary-backup [3] approaches select one replica as the primary server, and this replica handles all client requests and periodically synchronizes with one or more backup servers. In case of a failure of the primary, one of the backup servers takes over the service execution.

In general, primary-backup mechanisms are simpler, but yield longer recovery times as a backup must perform an explicit recovery algorithm to resume the server execution. On the other hand, active replication mechanisms have to deal with the overhead of handling several replicas and ensuring ordered delivery of messages to all replicas. Since the network capacity is very limited in mobile ad hoc networks, we prefer to leverage the primary-backup mechanism, which results in less traffic overhead.

Reliability in ad hoc networks has been mostly investigated at the networking layer to design reliable routing protocols [13, 26, 29]. Additionally, deterministic [7, 9, 19] and probabilistic [5, 15, 18] approaches for reliable broadcast and multicast have been investigated. Finally, reliable group communication [14] has also been studied. Unlike these projects, our work targets *service* reliability in ad hoc networks, a topic that was practically unexplored so far.

Numerous approaches have been studied in the past to achieve fault tolerance at the TCP level. Examples include FT-TCP [2], HYDRANET-FT [23], M-TCP [25], and ST-TCP [16]. It is hard, if not impossible, to apply these solutions to mobile ad hoc networks. The traditional end-to-end model of communication assumed by TCP does not work well in dynamic, multi-hop ad hoc networks [27]. Furthermore, it is difficult to introduce changes in TCP as network providers deny them. Trickles [24] is a TCP-like transport protocol that provides service continuation at the packet level and maintains the state only at the client side. However, this approach cannot be used in our case as we also need to capture and maintain the service state, not only the TCP connection state.

A solution to service continuity in ad hoc networks, called "follow-me" services, has been proposed in [10]. As the user moves through the network, services can migrate from node to node to maintain a seamless interaction with the client application. As such, these services can be considered a particular example of migratory services where the migration is triggered by the lack of connectivity between client applications and services. However, differently from migratory services, "follow-me" services assume to be capable of constantly predicting when a node is about to lose connectivity from another node and do not guarantee reliability against unpredictable node failures.

One.world [8] is similar to our work in the sense that both consider migration as a key mechanism to adapt to dynamic computing environments. Each application in *one.world* has at least one environment that contains tuples, application's components, and other nested environments. When needed, a migration can move a copy of an environment to another node. *One.world* also provides checkpointing support to capture the execution state of an environment tree and save it as a tuple. However, in *one.world*, checkpointing is mainly intended to resume an application after it has been dormant or after a failure, but not to provide reliability in a distributed client-service interaction.

7. CONCLUSIONS

This paper presented the design and implementation of a context-aware fault-tolerance mechanism that allows migratory services to survive crash failures and transient communication failures. We demonstrated the feasibility of our approach through a prototype implementation running on

smart phones and investigated its performance in large scale networks through simulations. Compared to other primary-backup approaches where the client node or a random neighbor of the primary are selected as backup nodes, our adaptive approach has succeeded in providing better reliability performance and lower network overhead.

Acknowledgments

This material is partly based upon work supported by the National Science Foundation under Grants No. CNS-0520033, CNS-0454081, IIS-0534520, and IIS-0714158. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Acharya and B. R. Badrinath. Checkpointing distributed applications on mobile computers. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS'94)*, pages 73–80. IEEE Computer Society Press, 1994.

[2] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. In *Proceedings of INFOCOM'01*, pages 329–337, 2001.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley, 1993.

[4] G. Cao and M. Singhal. Mutable checkpoints: A new checkpointing approach for mobile computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):157–172, 2001.

[5] R. Chandra, V. Ramasubramanian, and K. Birman. Anonymous Gossip: Improving Multicast Reliability in Mobile Ad-Hoc Networks. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 275–283, 2001.

[6] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[7] T. Gopalsamy, M. Singhal, D. Panda, and P. Sadayappan. A Reliable Multicast Algorithm for Mobile Ad Hoc Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 563–570, July 2002.

[8] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System Support for Pervasive Applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[9] S. K. Gupta and P. K. Srimani. An Adaptive Protocol for Reliable Multicast in Mobile Multi-hop Radio Networks. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, page 111, 1999.

[10] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Context Aware Session Management for Services in Ad Hoc Networks. In *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, pages 113–120, July 2005.

[11] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems. *The Computer Journal*, pages 475–494, 2004. The British Computer Society. Oxford University Press.

[12] B. Karp and H. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'00)*, pages 243–254, August 2000.

[13] R. Leung, J. Liu, E. Poon, A.-L. C. Chan, and B. Li. MP-DSR: A QoS-Aware Multi-Path Dynamic Source Routing Protocol for Wireless Ad-Hoc Networks. In *Proceedings of the 26th Annual IEEE Conference on Local Computer Networks (LCN'01)*, pages 132–141, 2001.

[14] J. Luo, P. Eugster, and J. Hubaux. Pilot: Probabilistic lightweight group communication system for ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):164–179, April 2004.

[15] J. Luo, P. T. Eugster, and J.-P. Hubaux. Route Driven Gossip: Probabilistic Reliable Multicast in Ad Hoc Networks. In *Proceedings of INFOCOM'03*, pages 2229–2239, March 30 - April 3 2003.

[16] M. Marwah, S. Mishra, and C. Fetzer. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, page 373, 2003.

[17] J. Nzouonta, N. Rajgure, G. Wang, and C. Borcea. VANET routing on city roads using real-time vehicular traffic information. Under Submission, December 2007.

[18] Öznur Özkasap, Z. Genç;, and E. Atsan. Epidemic-based approaches for reliable multicast in mobile ad hoc networks. *SIGOPS Oper. Syst. Rev.*, 40(3):73–79, 2006.

[19] E. Pagani. Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. *Mob. Netw. Appl.*, 4(3):175–192, 1999.

[20] C. Perkins and E. Royer. Ad-Hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, 1999.

[21] N. Ravi, C. Borcea, P. Kang, and L. Iftode. Portable Smart Messages for Ubiquitous Java-Enabled Devices. In *Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, pages 412–421. IEEE Computer Society, 2004.

[22] O. Riva, T. Nadeem, C. Borcea, and L. Iftode. Context-aware Migratory Services in Ad Hoc Networks. *IEEE Transactions on Mobile Computing*, 6(12):1313–1328, December 2007.

[23] G. Shenoy, S. K. Satapati, and R. Bettati. HYDRANET-FT: Network Support for Dependable Services. In *Proceedings of the the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, page 699, April 2000.

[24] A. Shieh, A. C. Myers, and E. G. Sirer. Trickles: A Stateless Network Stack for Improved Scalability, Resilience and Flexibility. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 175–188, May 2005.

[25] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection Migration for Service Continuity in the Internet. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, page 469, July 2002.

[26] J. Tang, G. Xue, and W. Zhang. Reliable routing in mobile ad hoc networks based on mobility prediction. In *Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS'04)*, pages 466–474. IEEE Computer Society, 2004.

[27] C. Wan, A. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA'02)*, pages 1–11, Atlanta, GA, September 2002.

[28] World Wide Web Consortium, Cambridge, MA, USA. *Efficient XML Interchange Measurements Note*, July 2007. W3C Working Draft.

[29] Z. Ye, S. V. Krishnamurthy, and S. K. Tripathi. A Framework for Reliable Routing in Mobile Ad Hoc Networks. In *Proceedings of INFOCOM'03*, pages 270–280, March 30 - April 3 2003.