DTN-RPC: Remote Procedure Calls for Disruption-Tolerant Networking

Artur Sterz*, Lars Baumgärtner*, Ragnar Mogk†, Mira Mezini†, Bernd Freisleben*†

*Dept. of Mathematics & Computer Science, Philipps-Universität Marburg, D-35032 Marburg, Germany
E-mail: {sterz, lbaumgaertner, freisleb}@informatik.uni-marburg.de
†Dept. of Computer Science / Electrical Engineering & Information Technology, TU Darmstadt, D-64293 Darmstadt, Germany
E-mail: {mogk, mezini}@st.informatik.tu-darmstadt.de; bernd.freisleben@maki.tu-darmstadt.de

***Abstract*—Remote Procedure Calls (RPCs) realize client-server interactions via a request-response message-passing protocol. They simplify distributed application programming by eliminating the need for explicitly having to code the details of a remote interaction. However, none of the existing RPC implementations are designed to work properly for Delay/Disruption-Tolerant Networking (DTN) where network connectivity is periodic, intermittent, and prone to disruptions. In this paper, we present *DTN-RPC*, a new approach to provide RPCs for DTN environments. *DTN-RPC* relies on (a) control and data channels to cope with potentially short contact durations in DTN where large amounts of data cannot be transmitted, (b) explicit and implicit modes to address remote servers, (c) Non-DTN and DTN transport protocols for issuing calls and receiving results, and (d) predicates that servers check to decide whether a procedure should be executed. The implementation of *DTN-RPC* is based on Serval, an open-source, disruption-tolerant wireless ad-hoc networking system. Our experimental results indicate that the measured CPU and network overheads for *DTN-RPC* are reasonably low so that it can be executed on smartphones or routers, and that the round-trip times and the number of successful RPCs are highly satisfactory in dynamic networks with unstable links.**

I. Introduction

The possibility of calling a procedure on a remote computer has been introduced to program client-server interactions in a procedural manner. *Remote Procedure Calls* (RPCs) [1] have proven to be useful in many distributed computing scenarios to simplify application programming by eliminating the need for explicitly having to code the details of remote interactions based on a request-response message-passing protocol. RPCs have been integrated into programming languages (e.g., Java RMI, Python RPyC, Distributed Ruby DRb-RPC, and Erlang RPC), dedicated applications (e.g., SAP RFC), and WWW protocols (e.g., XML-RPC, JSON-RPC, SOAP, Windows WCF, Google gRPC, Google Web Toolkit RPC).

However, none of the existing RPC implementations are designed to work properly for Delay/Disruption-Tolerant Networking (DTN) [2], [3] where network connectivity is periodic, intermittent, prone to disruptions, and a direct connection to a remote server might not exist. DTN scenarios with potentially large transmission delays as a result of either inadequate physical link properties or extended periods of network partitioning are common in natural disasters. For example, during the 2010 earthquake in Haiti, public and mobile telephone systems were destroyed or disturbed and could not be rebuilt or repaired for days[1]. An inoperative cellular communication infrastructure during the earthquake in New Zealand on November 14, 2016, created uncertainty about whether people were still in the affected areas[2]. Even in the absence of disasters, there are still regions, e.g., in India [4] and Australia [5], where no telecommunication infrastructure exists and where people cannot communicate using mobile devices. Whenever reliable end-to-end connectivity is not available, DTN can be used to sustain communications without requiring any conventional infrastructure.

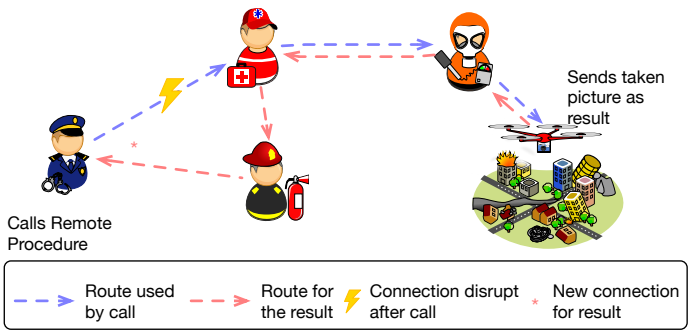


Fig. 1: Calling a remote procedure in a DTN disaster scenario.

Being able to use RPCs in these scenarios could provide great services for civilians and professional first responders. For example, quadcopters could offer a procedure that takes a picture with a mounted camera at a particular geographical location and returns it over the network. Then, rescuers could request an overview image via an RPC to a quadcopter while performing other tasks until the file arrives over a DTN connection using nodes of other rescue workers or citizens as relay nodes. This example is illustrated in Fig. 1, where the call takes the blue route, but the result arrives over the red route due to the connection loss illustrated by the yellow

[1]https://en.wikipedia.org/wiki/2010_Haiti_earthquake
[2]http://www.bbc.com/news/world-asia-37970775

ISBN 978-3-901882-94-4

lightning symbol. This might take longer, but without DTN the call could not be made at all.

In this paper, we present *DTN-RPC*, a new approach to provide RPCs for DTN environments. *DTN-RPC* relies on (a) control and data channels to cope with potentially short contact durations in DTN where it is impossible to transmit large amounts of data, (b) explicit and implicit modes for server addressing, (c) Non-DTN and DTN transport protocols for calls and results, and (d) predicates that servers can check to decide whether a procedure should be executed. Our open-source implementation of *DTN-RPC*[3] is based on Serval [6], [7], [8], [9], an open-source, disruption-tolerant wireless ad-hoc networking system. Our experimental results obtained within the network emulation framework CORE indicate that the measured CPU and network overheads for *DTN-RPC* are reasonably low so that *DTN-RPC* can be executed on smartphones or routers, and that the round-trip times and the number of successful RPCs are highly satisfactory in dynamically changing network topologies with unreliable connectivity.

The paper is organized as follows. Existing work on RPCs will be examined in Sec. II. The design of *DTN-RPC* will be presented in Sec. III. Implementation issues will be discussed in Sec. IV. Experimental results will be shown in Sec. V. Sec. VI concludes the paper and outlines areas of future work.

II. Related Work

Tu and Stewart [10] present a Java RPC framework where small data is replicated and sent over a second TCP connection to the server or back to the client. At the destination, a listener collects all arriving data on all connections, reassembles the original data, and passes it to the corresponding handler.

Stuedi et al. [11] increase the efficiency of RPCs in data centers by softening the userland and kernel separation in the network stack and by using remote direct memory access to minimize the overhead of network operations by performing them with less context switches and zero-copy network I/O.

Chen et al. [12] introduce memory regions where server and client exchange data to improve the efficiency of RPCs between virtual machines (VMs) on the same host computer. The proposed framework has three components: (a) a notification channel that informs the server about new calls and the client about arriving results, (b) a control channel that sends meta-data (e.g., the parameter count), and (c) a transfer channel that is responsible for transmitting data between server and client and putting the data in the predefined memory regions.

Shyam et al. [13] propose solutions for situations where an RPC server is not available. The first solution is a heartbeat server that observes whether the RPC server is operative. The second solution is that every node sends a health check message to the RPC server. Since these messages are typically smaller than an RPC request and no computations take place, the answer of the health check should arrive faster. If the answer does not arrive within a timeout that is smaller than the timeout for the RPC, the server is considered inoperative.

[3]https://github.com/adur1990/*DTN-RPC*

Reinhardt et al. [14] address the problem of providing RPCs in wireless sensor networks. In particular, the authors eliminate the need of conventional RPCs to send predefined data to predefined destinations, typically addressed by ports, by publishing descriptions of new sensors that can be used by other sensors or nodes dynamically.

Shi et al. [15] present a framework where mobile devices can offload jobs to other mobile devices. In scenarios where node mobility is high, only small tasks will be offloaded; otherwise larger jobs will be offloaded, too. To increase the number of offloaded jobs, every job is split into smaller tasks. Additionally, every node has to announce its capabilities, such as CPU capacity and available battery power. To offload a job, the framework compares the task requirements with the capabilities of the client and tries to find a server that satisfies the requirements better than the client. If no server is found, the job will be executed locally.

Chen et al. [16] propose a solution for offloading computations to ad-hoc cloudlets. A job is offloaded via an ad-hoc communication channel that is closed after the procedure has been called successfully. The result of the job can arrive (a) via an ad-hoc channel if server and client are in close proximity, (b) via a cellular network used when an ad-hoc connection is not possible, (c) via a WiFi access point, if available.

Zhang et al. [17] propose a solution for cloudlets with intermittent connectivity where parts of a job will be executed either locally or remotely. The decision which of both options is chosen is based on a probability that includes the cost of executing a task. Two cost factors are calculated: (a) the cost when the phase is executed locally, where, e.g., energy consumption is important, (b) the cost when the phase is executed in a cloudlet, where, e.g., available bandwidth is important. Based on this information, a Markov chain can be constructed and the optimal path can be found.

Lai et al. [18] propose an offloading algorithm for delay-tolerant mobile networks that increases the amount of offloaded data without increasing the transmission overhead or delay. The transfer channel is chosen based on the contact duration between two nodes and the available transmission protocols. Therefore, every node logs which neighbors are available. Based on the available neighbors, on the size of the data that is offloaded, and the estimated waiting time, a priority is computed. With these factors, a utility is calculated that denotes whether data should be offloaded using this particular channel or not.

To summarize, several of the related works address problems of RPCs in traditional networks, where links are either static or tasks are on the same machine, such as in VMs. Furthermore, direct memory access methods to reduce networking overhead cannot be used in a DTN environment, due to possibly untrustworthy nodes. Also, control mechanisms like heartbeats or duplicating data on multiple channels are no options for DTN. In the offloading approaches, the particular problems of RPCs in DTN are either not addressed or would require additional infrastructure, such as cell towers for 3G or LTE connectivity, or nodes with access to the Internet.

Our proposed *DTN-RPC* is designed to provide RPCs in DTN environments without requiring any additional infrastructure.

III. DTN-RPC

This section presents the design of *DTN-RPC*.

A. Fundamental Considerations

There are several differences between RPCs in traditional networks and RPCs in DTN.

In conventional RPC implementations, errors are handled, for example, if the connection between client and server is lost. In DTN, it is not certain whether a call even reaches its destination. Thus, errors in DTN can only be handled in a few situations, since error reports could just not arrive and the client would not notice that the call was not successful. The server, on the other hand, would have to spend computational overhead while trying to inform the client about the error. Furthermore, disruptions and poor connection quality make it impossible to support real-time communication or to guarantee a predefined quality of service in DTN.

Common RPCs are location transparent. For this purpose, stubs or proxy functions exist to handle communication via the network. In DTN, a call will explicitly be executed remotely, and it is expected that there will be networking overhead when executing a remote procedure.

In several RPC implementations, the client has to register at the server before calling a procedure. Since in DTN the address of a server is typically not known, client registration is not possible.

Traditional RPC servers either announce the procedures they offer or there exists a lookup service where clients can find information about which server offers which procedure. In DTN, server announcements might not reach or lookup services might not be available for clients when needed.

B. Control and Data Channels

DTN is often used in mobile mesh and ad-hoc networks where the network topology changes frequently. This can lead to short contact durations between nodes where it is impossible to transmit large amounts of data. Due to this restriction, two separate communication channels are introduced in *DTN-RPC*: the *control* and the *data* channel.

The *control* channel is responsible for transmitting meta-data, such as the procedure name and the parameters, from client to server, and possible results from server to client. The *control* channel supports two modes to address remote servers, *explicit* and *implicit* (*any* or *all*), as described below.

Explicit: If the address of a server is known and the server is reachable, *DTN-RPC* will choose the *explicit* mode and will try to establish an end-to-end connection to this specific server.

Implicit (any or all): If the address of a server is not known, but potential servers are reachable, both the *any* and *all* modes (summarized as the *implicit* mode) are used to broadcast a call. In the *any* mode, the client waits for exactly one response. This is helpful if it is known that servers exist that offer a particular procedure, but it does not matter which server responds. The first arriving response will be accepted. In the *all* mode, the client will wait for as many answers as possible until its internal timeout occurs. This is useful in scenarios where the quality of the results varies with the executing machine (e.g., GPU support, different algorithms), where different answers should be combined (e.g., to implement aggregate functions that return a value across all items in the results set), or is influenced by other factors such as geolocation (e.g., sensor readings, taking a picture).

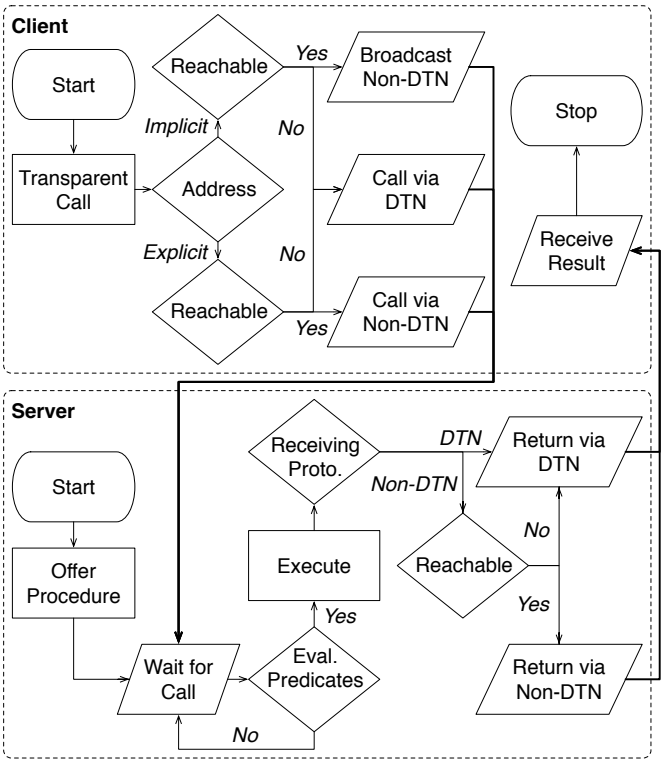


Fig. 2: *DTN-RPC* flowchart for client and server.

The payload of the *control* channel packets must not exceed the payload size of the underlying transport protocol to keep the data on the network as small as possible.

The *data* channel transports larger amounts of data from client to server and vice versa. It is used if a file is required as a parameter for a particular call. The transport of the payload in the *data* channel is always performed via DTN. The transport of the meta-data in the *control* channel is explained below.

C. Transparency

In both *explicit* and *implicit* addressing modes, the *control* channel of *DTN-RPC* supports *Non-DTN* and *DTN* transport protocols and automatically switches between them for performing a procedure call, as explained below.

Non-DTN vs. DTN: As illustrated in Fig. 2, if the server is reachable in the *explicit* mode, *DTN-RPC* will use a Non-DTN transport protocol to call the server. If the server is not reachable, the call will be issued using a DTN protocol.

After having called a remote procedure in the *explicit* mode, the client waits for the response using the same transport

protocol that was used to call the procedure. If the connection is interrupted, the client additionally waits for results that arrive via a DTN protocol.

After having successfully executed a received call, the server checks whether the explicit control channel on which the call was received via a Non-DTN protocol is still available, as shown in Fig. 2. If the channel is not available anymore, the result will be sent via a DTN protocol. The *DTN-RPC* server does not attempt to reestablish a Non-DTN connection, since it is unlikely that a reconnection is successful if one of the nodes has physically moved out of the network's reach. If the call was received via a DTN protocol, the server also uses a DTN protocol for its response.

Since the *implicit* modes use broadcast addresses to call procedures, a different transport protocol has to be used than in the *explicit* mode, because reliable point-to-point transport protocols like TCP do no support broadcast packets. Since a server availability check in a broadcast scenario would imply communication between multiple nodes, which would add additional delays, a call just gets broadcasted without any prior availability checks. If a timeout occurs and no result arrives, the call is performed via a DTN protocol.

Transparent: DTN-RPC is designed to automatically select the most suitable transport protocol in any given scenario. In the *transparent* transport method, both client and server are designed to make all the above dicussed decisions without any user interaction.

D. Offering and Executing Calls

To offer a remote procedure as shown in Fig. 2, two steps are required on the server: declaring and implementing a procedure. The first step of offering a remote procedure is that every procedure has to be declared as a prototype in an extra configuration file in order to tell the server which procedures are available for execution. The implementation of a procedure, which is the second step, has to be provided as an external executable written in any programming language.

The parameters of an incoming call are passed in the order they were received to the external program that then executes the procedure. After the procedure finishes, the result is returned to the server that marshals the result and prepares the result to send it back to the client.

Typically, the computational resources and the battery lifetimes of nodes in DTN are limited. To avoid the execution of calls that would consume too many resources with respect to the current state of a server, a server can decide whether a remote procedure should be accepted or not. For this purpose, we define particular predicates per server, such as thresholds for resource constraints (number of concurrent processes, remaining battery life etc.) or available (sensor) hardware like GPS. This is also shown in Fig. 2. The server checks whether defined predicates are satisfied. If at least one requirement is not met, the procedure will not be executed.

Furthermore, each call can provide its own requirements that also have to be checked by the server. For example, some calls should only be executed on non-moving nodes, or require special sensor hardware or extensive resources, such as disk space or RAM. Therefore, there is a two-stage predicate check per server: the first one is the general server acceptance check, and the second one is call-specific and evaluated after having passed the first check.

IV. Implementation

The implementation of *DTN-RPC* is based on the Serval Project [6], [7], [8]. Serval is centered around a suite of protocols designed to allow ad-hoc and infrastructure-independent communications. The Serval Mesh Protocols abstract from lower-layer protocols, such as IP, UDP, WiFi, packet radio or others. Serval's real-time packet-switched protocol is the Mesh Datagram Protocol (MDP), which can be compared to UDP/IP, but uses SIDs (Subscriber ID, the public key of an asymmetric elliptic curve key pair) instead of IP addresses, and includes encryption, authentication and integrity features by default. To route packets, MDP uses a protocol inspired by OLSR [19] and B.A.T.M.A.N. [20] for both node discovery and maintaining a routing table, which facilitates multi-hop routing of packets. On top of MDP, the Mesh Streaming Protocol (MSP) provides reliable data streaming, similar to TCP. Finally, Rhizome is a simple store-and-forward protocol defining files as *bundles*. Intended as the DTN protocol of Serval, Rhizome uses an epidemic routing protocol to transmit files hop-by-hop from source to destination. Rhizome is purposely agnostic of the transport protocols below it, requires no routing table and focuses on single-hop communications, with multi-hop communications emerging as a natural consequence of *bundles* replicating among nodes. *DTN-RPC* uses MDP, MSP, and Rhizome to handle different situations and addressing modes.

We have conducted an in-depth experimental evaluation of Serval's DTN aspects for various network setups and usage patterns in our previous work [9]. The results have indicated that Serval is capable of handling extreme conditions such as saturated networks or many-hop transmissions in a satisfactory manner. It has also been shown that Serval works well in realistic scenarios, where the topology changes over time and users have different requirements. Thus, Serval is an elaborate and ready-to-use software for DTN and mesh networks.

For programmers, an API is offered that can be used to develop programs using the *DTN-RPC* library to execute procedures on remote devices in DTN environments.

A. Calling a Remote Procedure Transparently

To call a remote procedure transparently, a single function is required that is part of the offered *DTN-RPC* API. This function has five parameters: the server address, the name of the called remote procedure, the number of parameters of the procedure, the parameters themselves and the execution requirements discussed in Section III-D. The mode to be used is determined by the first parameter of this API function call.

1) Explicit: If the parameter is a valid address, the remote procedure will be called explicitly, i.e., the call will be issued via Serval's MSP, if the server is available. A routing table is built in an ad-hoc manner. If the address of the server can be

found in this routing table, this particular server is reachable. While waiting for the result, the client checks periodically whether the connection is still alive. If the connection terminates, the client starts a Rhizome DTN listener.

2) Implicit: The modes *any* and *all* are used if the address is the *ANY* address provided by Serval for *any* or the broadcast address for *all*. Since Serval's MSP supports point-to-point communication only, it is not possible to send data to the broadcast address. Therefore, *any* and *all* use Serval's MDP.

Since a reachability test is not possible for broadcast packets, the procedure will be called without any prior checks. Since delivery is uncertain, the client sends a call every second until at least one server responds with an acknowledgement or a timeout occurs. If an acknowledgment arrives, the threshold for the timeout is increased. Only if the new timeout occurs, the client will additionally start a Rhizome DTN listener and wait for the result via DTN.

The difference between the modes *any* and *all* is the number of results. In the first case, the client stops listening as soon as the first result arrives. In the second case, the client waits for as many results as possible, but at least for one.

B. Returning the Result Transparently

While executing the called procedure, the server does not check periodically whether the client is still reachable. Instead, this check is done once when the response is ready to be sent. If the call arrived via MSP or MDP, but the connection is broken or the client is not reachable, sending will fail and the server will send the result via Rhizome.

V. Experimental Evaluation

In this section, we present an experimental evaluation of *DTN-RPC* for different network topologies and in various configurations. Due to the lack of comparable RPC implementations that can handle disruptive networks, *DTN-RPC* is not compared against other approaches. A comparison with widespread software solutions such as JSON-RPC or SOAP would be unfair, since they would fail each time the network connection is lost.

A. Test Setup

Our evaluation of *DTN-RPC* is based on the open source network emulation framework CORE[4]. Compared to protocol simulations, CORE can run *DTN-RPC* without modifications in a more realistic Linux environment. All tests are performed on a 64-core AMD Opteron 6376 CPU with 256 Gigabyte RAM, emulating up to 64 virtual nodes at the same time.

1) Measurements: Standard Unix tools are used to measure system properties with a time resolution of one second. For CPU statistics, *pidstat*[5] is used, and the Serval and *DTN-RPC* processes are monitored from within a node. Network usage is measured from within the nodes on every network interface for Serval and *DTN-RPC* using a custom Python script based on *libpcap*[6]. To monitor the behavior of *DTN-RPC*, metrics such as call times, round-trip times, and logging functions were implemented and integrated into the binary.

TABLE I: Topologies

Name	# Nodes	Description
Hub	28	All nodes connected to each other
Chained	32	Pair-wise connected
Islands	64	Partitioned islands with dynamic links in between

2) Network Topologies: Three network topologies are considered, as shown in Table I.

a) Hub: The *Hub* topology connects 28 nodes with each other so that every node is one hop away from all other nodes. As shown in our previous work [9], the Hub topology is challenging for Serval and thus also for *DTN-RPC* due to the high number of direct neighbors, all using bandwidth and flooding each other with status information. Therefore, the *Hub* topology helps to investigate whether *DTN-RPC* can handle RPCs when the network is under heavy load.

b) Chained: The *Chained* topology consists of a chain of 32 nodes, 31 hops from the first to the last node. Typically, network connections over the Internet require less than 16 hops. In a DTN mesh network, more hops might be needed for messages to reach their destination.

c) Islands: The *Islands* topology represents a partitioned, dynamic network with 64 nodes. At the beginning, there are 4 islands each containing 16 nodes. The 16 nodes per island are connected randomly with each other, creating an ad-hoc mesh network. Then, four different behaviors can occur randomly every 60 seconds: two islands are connected, two connected islands are disconnected, all islands are connected or all islands are disconnected resulting in the original state.

3) Network Connections: DTN-RPC adds a new layer of abstraction to the Serval networking stack. Although Serval can cope with several degraded networking scenarios, *DTN-RPC* is only evaluated in situations where network connections are completely lost, because this is the most challenging situation in DTN. Network degradations and bandwidth limitations would only lead to higher delays, but not break *DTN-RPC* itself.

4) Test Sets and Modes: The remote procedure used in our tests implements a simple echo service. It is called with three different test sets: (a) *0MB*, where no file is used; (b) *1MB*, where a file of 1 megabyte is transmitted; (c) *100MB*, where a file of 100 megabyte is sent.

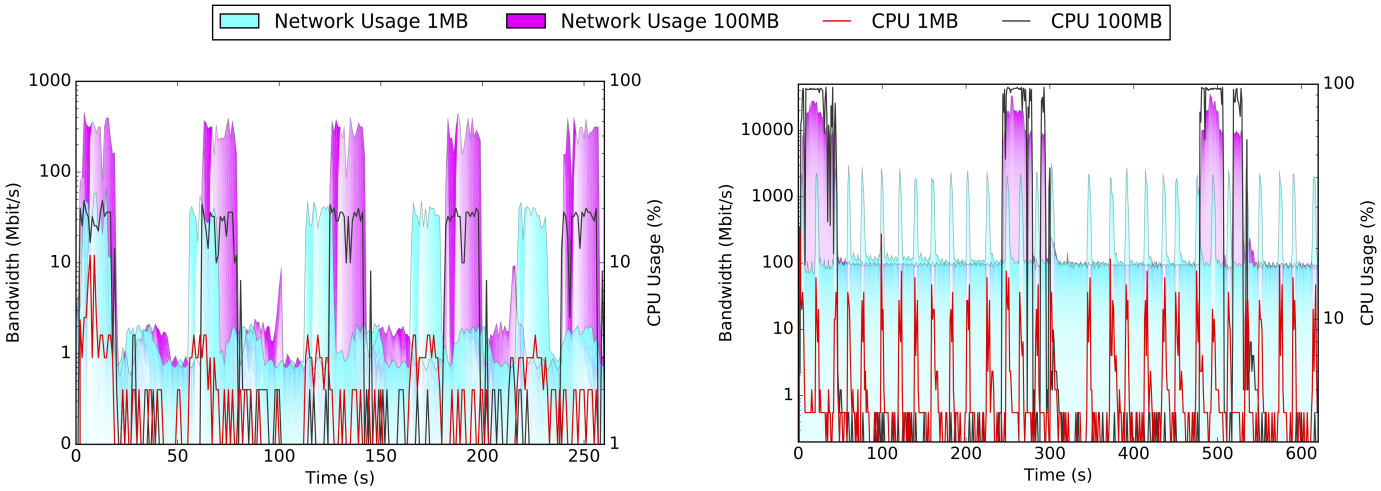
Additionally, all tests are executed in 10 different modes: *explicit*, *any* and *all* via Rhizome; *explicit*, *any* and *all* via MDP; *explicit*, *any* and *all* transparently and *explicit* via MSP.

5) Servers: Since the successful execution of remote procedures in DTN depends on the number and distribution of servers, every test in *Hub* and *Islands* is executed twice, first with 5% of the nodes as servers and second with 50%. In *Chained*, the goal is to determine how *DTN-RPC* performs if

[4]https://www.nrl.navy.mil/itd/ncs/products/core

[5]http://sebastien.godard.pagesperso-orange.fr

[6]http://www.tcpdump.org



(a) *Chained* topology

(b) *Hub* topology

Fig. 3: Stacked bandwidth usage for *1MB* and *100MB* and maximum CPU usage for *1MB* and *100MB* in different topologies.

the call has to travel a long distance. Thus, only one server and one client at the opposite ends of the chain are needed.

In each test setup, the procedure is called 30 times to get reliable results. The acknowledgement from the server has to arrive within 30 seconds on the explicit channel. After the acknowledgement, the client waits an additional 90 seconds for the result. If within these 90 seconds no results arrived, the procedure is called via DTN, which has an additional 90 seconds to finish. After the client has received the result or all timeouts are reached, the next procedure will be called.

Since our evaluation is concerned with the overhead and the performance of *DTN-RPC*, the possibility of *DTN-RPC* to perform predicate checks to decide whether a remote procedure should be accepted has been disabled in our experiments.

B. Fundamental Properties

In *Hub* where each node is a single hop away from all other nodes and Serval uses broadcast packets to announce meta-data, each node produces a flood of data that is sent to all neighbors. Thus, both the CPU usage and the network load in *Hub* are always higher than in the corresponding tests in *Chained* or *Islands*, due to the high number of direct neighbors. Furthermore, *DTN-RPC* does not only use the API, but also the networking stack and the communication mechanisms provided by Serval. Thus, *DTN-RPC* cannot be measured separately, but only together with other Serval traffic.

Similar to the network usage, the CPU utilization has to be measured not only for *DTN-RPC*, but also for Serval running on a node. The evaluation of the CPU usage shows that the CPU consumption of *DTN-RPC* is negligible with about 1% in heavy load situations. However, the Serval process has a higher CPU usage, since Rhizome computes a hash for each file sent. The larger the file, the more time-consuming the hash computation becomes. *DTN-RPC*, on the other hand, is independent of file sizes, because it simply issues a call to the Rhizome API, which leads to the described 1% CPU utilization increase in the *DTN-RPC* process. Therefore, since the CPU utilization is dominated by Rhizome, in the experiments below it is always based on the Serval process.

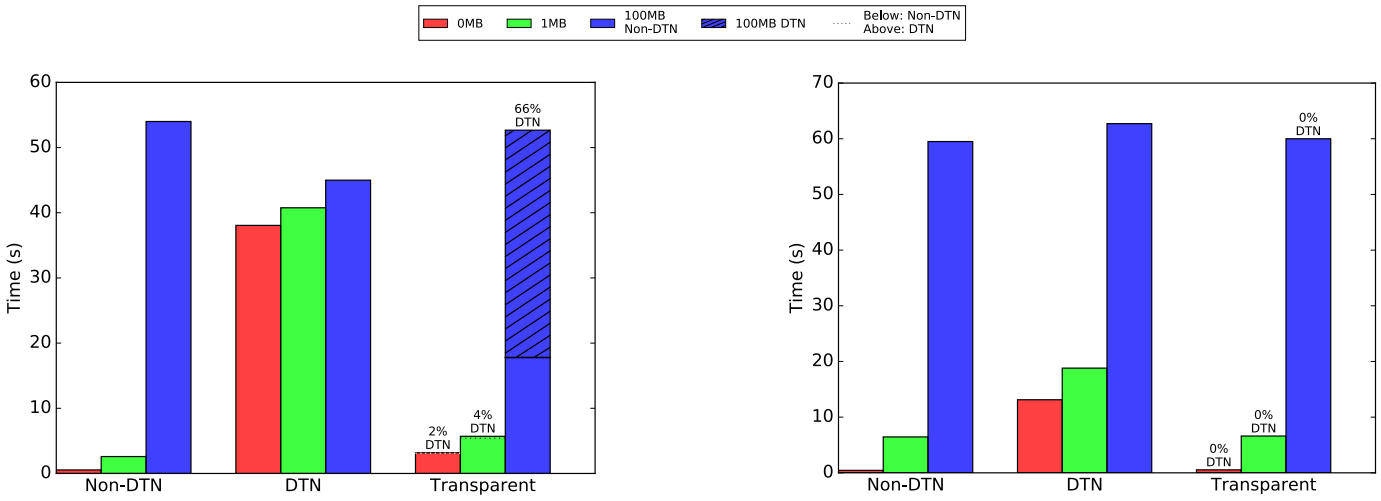
C. Network Performance

For the *0MB* tests in the *Chained* topology, the overall network load averages at about 2 Mbits per second for each of the three transport protocols (MDP, MSP and Rhizome). This is true for all three modes, *explicit*, *any* and *all*. Since *DTN-RPC* uses only a single packet for calling the remote procedure and returning the result in *0MB*, these packets get lost in the overall network load that is produced by Serval exchanging meta-data and therefore not plotted in Fig. 3.

During the *1MB* and *100MB* test sets, the network load increases up to 70 Mbit/s for *1MB* and up to 500 Mbit/s for *100MB*, as indicated by the blue and red graph of Fig. 3a, in which the stacked bandwidth for all network interfaces together with the CPU usage in a logarithmic scale for 5 calls with the *100MB* test set and 30 calls with the *1MB* test set is shown. In the *1MB* and *100MB* calls, a file always has to be transmitted via the Rhizome DTN for calling the remote procedure and receiving the result. The difference between the *1MB* and *100MB* calls is due to the different file sizes.

The *Hub* topology shows a similar behavior, as illustrated by Fig. 3b, where the stacked bandwidth for all network interfaces together with the CPU usage in a logarithmic scale for 3 calls with the *100MB* test set and 30 calls with the *1MB* test set is shown. The main difference is that the *Hub* topology suffers from the problems discussed in Sec. V-B. The overall network usage for the *1MB* test sets exceeds 1,000 Mbit/s (blue graph) and 10,000 Mbit/s for the *100MB* test sets (red graph).

Comparing the bandwidth consumption to our previous results [9], *DTN-RPC* does not add any measurable network traffic to the traffic produced by Serval, and thus can handle scenarios where the network has a high bandwidth usage well.



(a) *Chained* topology

(b) *Hub* topology

Fig. 4: Round trip times in different topologies.

D. CPU Usage

As shown in Figures 3a and 3b, CPU usage highly correlates with network usage. Since CPU usage in the *0MB* tests does not exceed 1% after the initial discovery phase, it is not plotted in Figures 3a and 3b. For the *1MB* tests, the maximum is at about 2% up to 3% (red line) and up to 20% for the *100MB* tests (black line) in the *Chained* topology.

In the *Hub* topology, the behavior is comparable to the *Chained* topology, with the difference that the CPU usage is generally higher. In the *1MB* tests, the CPU usage increases up to about 10% and for the *100MB* tests up to 90% during the sending phase. This relatively high CPU consumption happens only while a hash of a file is computed and the file is inserted into the Rhizome store, and thus only during a relatively short time period. As already mentioned, the CPU usage of *DTN-RPC* does not exceed 1%.

E. Round Trip Times

To measure the round-trip times (RTTs), only the *Chained* and *Hub* topologies are considered, since the *Islands* topology would not give any credible results due to the random merging and separation of the islands. RTT is only used to indicate the time that is needed to transmit the payload through the network to be sure no additional delays are introduced by *DTN-RPC*. The execution of a procedure typically takes longer to finish than the implemented echo service.

As shown in Fig. 4a, the *0MB* tests in *Chained* called by MDP or MSP (i.e., Non-DTN) are executed within a second. As the files grow, the RTT increases.

In the DTN tests, the RTTs are similar, regardless of the file size. Due to the fact that in DTN the control channel as well as the data channel are transferred via Rhizome, both server and client have to wait for two files. Therefore, all tests take about 40 seconds.

Transparent calls are slower than the calls via MDP or MSP for the *0MB* and *1MB* tests. Some of the calls are issued via MDP or MSP, while others are executed via Rhizome, as explained in Section III-C. The illustrated RTTs are averaged over 30 calls, including the slower Rhizome calls. Furthermore, the time it takes to wait until the transport protocol will be switched is also part of the RTT. Therefore, the *transparent* tests are slower than the corresponding *explicit* tests, but faster than the DTN tests. Since all *100MB* tests are issued using Rhizome and the switch time is included in the RTT, the time it takes for finishing is higher than for MDP or MSP.

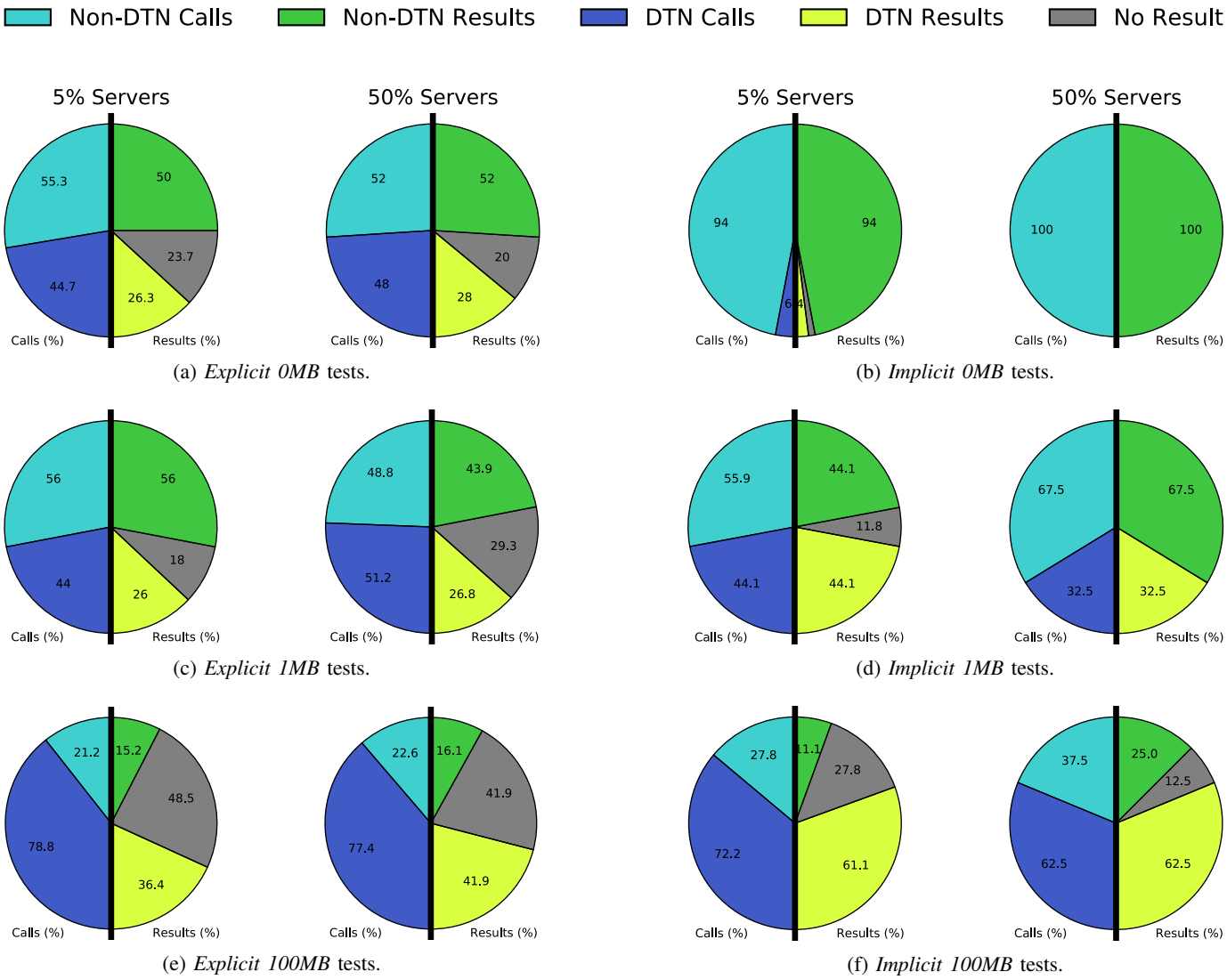
As shown in Fig. 4b, the RTTs for tests in *Hub* do not differ much from the tests in *Chained*. The only difference is that the *0MB* and *1MB* tests are faster in *Hub*, because all nodes are only one hop away from each other.

To summarize, *DTN-RPC* can execute remote procedures satisfactorily fast. The fallback method using Rhizome is slower, but still can get a result back to the client within an acceptable time, even if the files are large.

F. Transparency Behavior

In this section, we examine how *DTN-RPC* behaves in the dynamic *Islands* topology with different numbers of available servers. The figures below show how many of a total of 30 procedures are called using Non-DTN or DTN, respectively, in terms of percentage values. The left half of the pie charts represents outgoing calls and the right half incoming results.

Since the *Islands* topology consists of 4 islands with 16 nodes that merge and separate over time, it is possible that not all results arrive within 210 seconds at the client (see Sec. V-A5) if the call was issued in *explicit* mode, especially in tests with only 5% servers. Additionally, as the file size increases, the transmission time increases too, and the number of successful calls decreases as expected, as indicated by Fig. 5a, Fig. 5c, and Fig 5e. Furthermore, it is evident that some of the results arrive via MDP or MSP (i.e., Non-DTN), others only via Rhizome (i.e., DTN). There are two reasons. First, it is possible that a call is issued successfully using MSP,



(a) *Explicit 0MB* tests.

(b) *Implicit 0MB* tests.

(c) *Explicit 1MB* tests.

(d) *Implicit 1MB* tests.

(e) *Explicit 100MB* tests.

(f) *Implicit 100MB* tests.

Fig. 5: Percentages of procedures called and results returned via Non-DTN and DTN for *100MB* in the *Islands* topology.

but the route from the server to the client gets lost because the islands have separated. Then, the result is sent via Rhizome and arrives after the islands have merged again. Second, the client cannot establish a connection to the server at all, because the islands are not connected. The procedure will be called using Rhizome and the client will wait via Rhizome for the result. Even if some results do not arrive in the *explicit* mode, the DTN protocol helps to improve the number of successful calls, as shown in Fig. 5. 41.9% of the results in the *explicit* tests with the *100MB* test set with 50% of the nodes as servers arrive via Rhizome, and in 41.9% of the tests, no result arrives. In the *implicit* tests with the *100MB* test set with only 5% of the nodes as servers, 61.1% of the results arrive via Rhizome, and only 27.8% of the results do not arrive at all.

Figures 5b, 5d and 5f show *implicit* tests in the *Islands* topology for three different file sizes with different numbers of servers. It is evident that the *implicit* mode increases the number of successful calls in every situation compared to the *explicit* tests. Due to the dynamically changing *Islands* topology and the relatively short contact durations, it is still possible that not all results arrive in *100MB*. For the *explicit* calls, the more servers are available, the more results arrive.

The number of missing results can be decreased if the contact duration is increased or the waiting time for results is increased. Furthermore, more elaborate remote procedures require a lot more time to finish than the simple echo service used in our evaluation. Therefore, the waiting time for results of up to 210 seconds in our experiments should be increased in production environments, since it might be possible that a result arrives after hours at the client via DTN.

To summarize, the *transparent* mode helps to improve the probability of receiving results in dynamically changing network topologies like *Islands*. Furthermore, the *transparent* mode can deliver results where a traditional RPC would

not lead to any response due missing network connections. Finally, if the waiting time for results is adequately large, the probability of receiving results increases, because when a DTN protocol is used, results do not get lost, but simply are not transmitted via a direct connection to the receiving node. Therefore, given sufficient time, results will always reach their destinations.

VI. Conclusion

In this paper, we have presented *DTN-RPC*, a new approach to provide RPCs for DTN environments. *DTN-RPC* relies on (a) control and data channels to cope with potentially short contact durations in DTN where it is impossible to transmit large amounts of data, (b) explicit and implicit modes to address remote servers, (c) Non-DTN and DTN transport protocols for issuing calls and receiving results, and (d) predicates that servers check to decide whether a procedure should be executed. The implementation of *DTN-RPC* is based on Serval, an open-source, disruption-tolerant wireless ad-hoc networking system. Our experimental results have indicated that the measured CPU and network overheads for *DTN-RPC* are reasonably low so that *DTN-RPC* can be executed on smartphones or routers, and that the round-trip times and the number of successful RPCs are highly satisfactory in dynamically changing network topologies with unstable links. Thus, *DTN-RPC* adds remote computing capabilities in the form of RPCs to DTN. These can, for example, greatly improve the tools available for professional responders during emergencies by utilizing low-power mobile devices that can offload tasks, such as requests for aerial overview images or for face recognition based comparisons to search for missing people. Furthermore, CPU-intensive tasks such as reconstruction of 3D models for replication of spare parts in the field [21] can be delegated off-the-grid to more powerful participants in the area.

There are several areas for future work. First, *DTN-RPC* has been tested and evaluated using emulated networks. We plan to perform tests with smartphones to get a better view on the real-world performance of *DTN-RPC* and a realistic evaluation of its energy consumption. Second, since the Non-DTN transport protocols produced satisfactorily results in the *Chained* and *Hub* topologies, *DTN-RPC* should be evaluated without relying on the strict differentiation between control and data channels. Finally, although it is relatively difficult to implement error handling and acknowledgement mechanisms, our evaluation has shown that this is not impossible. Thus, an acknowledgement system should be implemented for the *any* mode to inform other servers that the execution has already started.

Acknowledgment

This work is funded by the LOEWE initiative (Hessen, Germany) within the NICER project and the DFG as part of the CRC 1053 MAKI.

References

[1] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[2] K. Fall, "A delay-tolerant network architecture for challenged internets," in *2003 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03. ACM, 2003, pp. 27–34.

[3] A. McMahon and S. Farrell, "Delay- and disruption-tolerant networking," *IEEE Internet Computing*, vol. 13, pp. 82–87, 2009.

[4] M. Kawecki and R. O. Schoeneich, "Mobility-based routing algorithm in delay tolerant networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2016, no. 1, pp. 1–9, 2016.

[5] P. Gardner-Stephen and S. Palaniswamy, "Serval mesh software - wifi multi model management," in *Proceedings of the 1st International Conference on Wireless Technologies for Humanitarian Relief*, ser. ACWR '11. ACM, 2011, pp. 71–77.

[6] P. Gardner-Stephen, R. Challans, J. Lakeman, A. Bettison, D. Gardner-Stephen, and M. Lloyd, "The Serval Mesh: A platform for resilient communications in disaster & crisis," in *IEEE Global Humanitarian Technology Conference (GHTC)*. IEEE, 2013, pp. 162–166.

[7] P. Gardner-Stephen, A. Bettison, R. Challans, and J. Lakeman, "The rational behind the Serval network layer for resilient communications," *Journal of Computer Science*, vol. 9, no. 12, p. 1680, 2013.

[8] P. Gardner-Stephen, J. Lakeman, R. Challans, C. Wallis, A. Stulman, and Y. Haddad, "MeshMS: Ad hoc data transfer within a mesh network," *International Journal of Communications, Network and System Sciences*, vol. 8, no. 5, pp. 496–504, 2012.

[9] L. Baumgärtner, P. Gardner-Stephen, P. Graubner, J. Lakeman, J. Höchst, P. Lampe, N. Schmidt, S. Schulz, A. Sterz, and B. Freisleben, "An experimental evaluation of delay-tolerant networking with Serval," in *IEEE Global Humanitarian Technology Conference (GHTC '16)*. IEEE, 2016, pp. 70–79.

[10] J. Tu and C. Stewart, "Replication for predictability in a Java RPC framework," in *2015 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2015, pp. 163–164.

[11] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "DaRPC: Data center RPC," in *ACM Symposium on Cloud Computing*, ser. SOCC '14. ACM, 2014, pp. 15:1–15:13.

[12] H. Chen, L. Shi, J. Sun, K. Li, and L. He, "A fast RPC system for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1267–1276, 2013.

[13] N. Shyam, C. Harmer, and K. Beck, "Managing remote procedure calls when a server is unavailable," May 2011, US Patent App. 12/610,049. [Online]. Available: https://www.google.ch/patents/US20110107358

[14] A. Reinhardt, P. S. Mogre, and R. Steinmetz, "Lightweight remote procedure calls for wireless sensor and actuator networks," in *, 2011 IEEE International Conference on Pervasive Computing and Communications Workshops*, 2011, pp. 172–177.

[15] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *13th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '12. ACM, 2012, pp. 145–154.

[16] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: architecture and service modes," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 18–24, 2015.

[17] Y. Zhang, D. Niyato, and P. Wang, "Offloading in mobile cloudlet systems with intermittent connectivity," *IEEE Transactions on Mobile Computing*, vol. 14, no. 12, pp. 2516–2529, Dec. 2015.

[18] Y. Lai, X. Gao, M. Liao, J. Xie, Z. Lin, and H. Zhang, "Data gathering and offloading in delay tolerant mobile networks," *Wireless Networks*, vol. 22, no. 3, pp. 959–973, 2016.

[19] T. Clausen and P. Jacquet, "Optimized link state routing protocol (OLSR)," Internet Requests for Comments, RFC Editor, RFC 3626, 2003.

[20] D. Johnson, N. Ntlatlapa, and C. Aichele, "A simple pragmatic approach to mesh routing using B.A.T.M.A.N." in *2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries*, 2008.

[21] J. Schöning and G. Heidemann, "Image based spare parts reconstruction for repairing vital infrastructure after disasters," in *IEEE Global Humanitarian Technology Conference (GHTC '16)*. IEEE, 2016, pp. 225–232.