MobiStreams: A Reliable Distributed Stream Processing System for Mobile Devices

Huayong Wang
MIT
Massachusetts, USA
huayongw@smart.mit.edu

Li-Shiuan Peh
MIT
Massachusetts, USA
peh@csail.mit.edu

Abstract*—Multi-core phones are now pervasive. Yet, existing applications rely predominantly on a client-server computing paradigm, using phones only as thin clients, sending sensed information via the cellular network to servers for processing. This makes the cellular network the bottleneck, limiting overall application performance. In this paper, we propose MobiStreams, a Distributed Stream Processing System (DSPS) that runs directly on smartphones. MobiStreams can offload computing from remote servers to local phones and thus alleviate the pressure on the cellular network. Implementing DSPS on smartphones faces significant challenges: 1) multiple phones can readily fail simultaneously, and 2) the phones' ad-hoc WiFi network has low bandwidth. MobiStreams tackles these challenges through two new techniques: 1) token-triggered checkpointing, and 2) broadcast-based checkpointing. Our evaluations driven by two real world applications deployed in the US and Singapore show that migrating from a server platform to a smartphone platform eliminates the cellular network bottleneck, leading to 0.78∼42.6X throughput increase and 10%∼94.8% latency decrease. Also, MobiStreams' fault tolerance scheme increases throughput by 230% and reduces latency by 40% vs. prior state-of-the-art fault-tolerant DSPSs.

***Keywords*-stream computing; mobile computing; reliability;**

I. Introduction

Smartphones have become powerful. Nowadays, it is common for a smartphone to have a quad-core CPU of 1.5∼2.0GHz, such as Samsung Galaxy S 4, Lenovo K860i and Huawei Ascend P6. Besides, octa-core 2GHz CPUs for mobile phones will come to market soon [1]. These phones' computation capability is comparable to that of servers 7 years ago in terms of the total processor frequency on a single chip (Intel's first quad-core CPU was released in 2006). Meanwhile, smartphone penetration rates in the U.S. and Singapore have reached 60% and 90% [2, 3]. Yet, existing systems and applications typically use smartphones as data-collecting devices or thin clients, transferring all sensed data from the phones to back-end servers for computing. These systems and applications have several shortcomings stemming from the client-server computing paradigm which relies on the slow, overloaded, costly cellular network. The 3G cellular network is already at capacity in many countries. Furthermore, it is projected that global monthly mobile data traffic will increase 18-fold from 597 petabytes in 2011 to 10.8 exabytes in 2016 [4]. While the 4G/LTE cellular network can increase the bandwidth by 20X, the projected demand will still exceed capacity in 2016.

A popular client-server computing paradigm used for real-time data analytics today is Distributed Stream Processing Systems (DSPSs), such as IBM's InfoSphere Streams [5], Yahoo's S4 [6] and Streambase [7]. DSPSs have been applied to many industries, ranging from transportation to healthcare, energy management to finance [8]. DSPSs improve software development productivity by hiding the complexity of fault tolerance, workload balance and network management while providing the scalable computing power of a cluster of servers. A DSPS reads in live data streams from end user devices (e.g. smartphones or sensors) and distributes processing tasks to multiple servers in a data center. The servers process these data streams and return the computed results back to the user devices. The underlying hardware platform for DSPSs is thus usually a cluster comprising reliable servers interconnected by high-speed Ethernet. In this paper, we propose harnessing the increasingly powerful and pervasive smartphones as a distributed computing platform, processing sensed data directly on the phones themselves, thus avoiding the shortcomings of the client-server computing paradigm discussed. Specifically, we propose MobiStreams, a DSPS that runs directly on smartphones. We show that MobiStreams can increase the throughput of our two example applications by 0.78∼42.6X and reduce the latency by 10%∼ 94.8% vs. traditional server-based DSPSs (Section IV-A).

Porting a DSPS from reliable servers interconnected by high-bandwidth Ethernet to a collection of smartphones interconnected by ad-hoc WiFi brings about immense fault tolerance challenges. Smartphones can readily fail due to limited battery, mobility and/or the unreliable wireless network. Previously proposed DSPS fault tolerance schemes [9–15] are all for servers. They do not work well on smartphones for two reasons. First, the failure model of a smartphone platform is quite different from the assumptions made by previous work, which assumes small-scale (usually single node) failures and is based on the belief that such 1-safety guarantee is sufficient for most real-world applications [16]. However, in a smartphone platform, it is common that several phones fail simultaneously. For example, multiple phone users may move out of a communication region

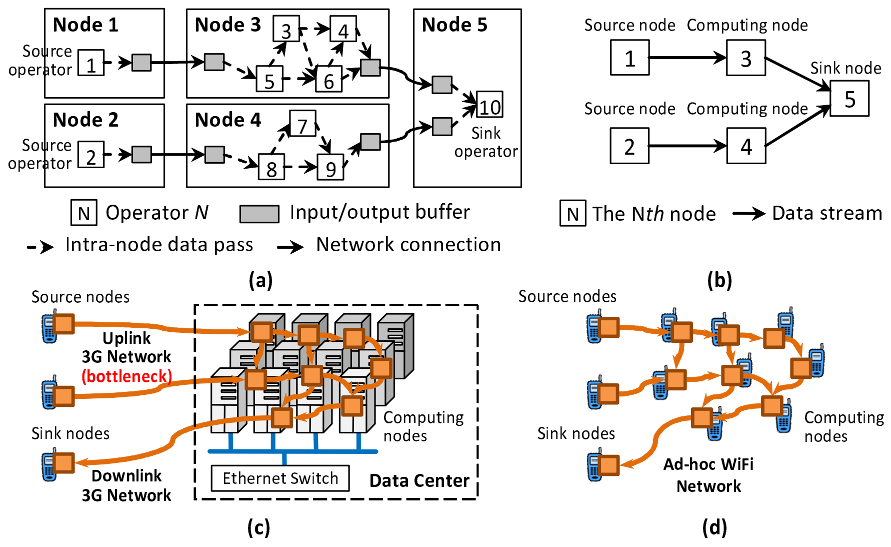


Figure 1. An example of DSPS. (a) DSPS represented by operators and a query network. (b) DSPS represented by nodes and a high-level query network. (c) A DSPS deployment on servers, using smartphones as data-collecting devices. (d) A DSPS deployment on smartphones.

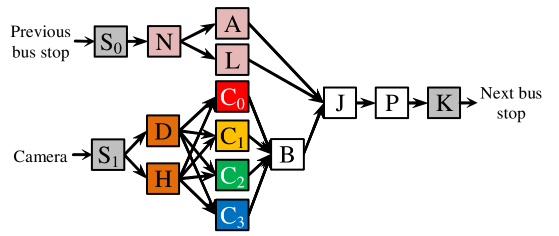
at the same time, losing WiFi connectivity. Second, prior fault-tolerance schemes impose substantial overhead on the performance of DSPS applications, as they require sending a large amount of extra data over the network, while the phone-to-phone ad-hoc WiFi network in a smartphone platform has limited bandwidth. Therefore, porting the existing fault tolerance schemes naively to smartphone platforms leads to poor resilience and performance. An ideal DSPS should be resilient to many faults, while adding little runtime overhead for fault tolerance.

MobiStreams is a checkpoint-based reliable DSPS tailored for smartphones. It can overcome large scale burst failures while providing good performance. MobiStreams comprises two new approaches to reduce the checkpointing overhead. 1) **Token-triggered checkpointing:** MobiStreams introduces tokens to coordinate the checkpointing activities. The tokens are generated by source operators. The tokens trickle down the stream graph, triggering each operator to checkpoint its own state. Token-triggered checkpointing avoids the redundant data saving in prior fault tolerance schemes and hence reduces the amount of the data to be saved during a checkpoint. 2) **Broadcast-based checkpointing:** When sending checkpoint data over the network, MobiStreams splits up the data transmission into multiple phases. In the first few phases, it uses unreliable UDP broadcasts. The UDP broadcasting avoids redundant data transmission over the network and reduces network overhead.

II. Background and Motivation

A. Distributed Stream Processing System

A DSPS consists of operators and connections between operators. Fig. 1.a illustrates a DSPS and a stream application running on the DSPS. The application contains ten operators. Each operator is a piece of program code executed repeatedly to process its input data. Whenever an operator finishes processing a unit of input data, it produces output data and sends them to the next operator in the graph. Each unit of the data passed between two operators is called a tuple. The tuples sent in a connection between two operators form a data stream. A directed acyclic graph, termed query network, specifies the producer-consumer relations between operators. The query network starts from source operators, which are responsible for fetching data from external data sensors, and terminates at sink operators, which publish results to the end users. Multiple operators can run on the same node, and a group of operators on a node can be treated as a single super operator. Without loss of generality, we assume each node has only one operator. Consequently, an operator is the smallest unit of work that can be checkpointed and recovered independently. The structure of a stream application can be represented at a higher level based on the interaction between the nodes, as shown in Fig. 1.b. The nodes that run source and sink operators are called source and sink nodes respectively. The nodes that run other operators in the DSPS are called computing nodes. In Fig. 1.b, node 1 is the upstream node of node 3, while node 5 is node 3's downstream node. Fig. 1.c illustrates the deployment of a conventional server-based DSPS. Smartphones are only used as data-collecting devices. The uplink cellular network is usually a bottleneck because of its low bandwidth. MobiStreams aims to offload most of the computing onto smartphones, as shown in Fig. 1.d.



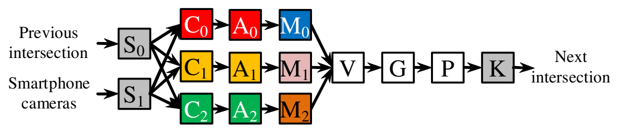
S₀: data from previous bus stop, **N**: noise filter, **A**: prediction model for bus arrival time, **L**: prediction model for alighting passengers, **S₁**: camera data source, **D**: dispatcher, **H**: motion detection (passerby filter), **C**: counter (counting people in images), **B**: predication model for boarding passengers, **J**: join, **P**: prediction model for bus capacity, **K**: sink (to next bus stop).

Figure 2. The DSPS at each bus stop of the Bus Capacity Prediction application. The operators with the same color are on the same node.

B. Motivating Applications

MobiStreams is motivated by two real-world applications in transportation that have been deployed at two college campuses in the US and Singapore. The first application is Bus Capacity Prediction (BCP), which has been deployed along a campus shuttle bus route in Singapore. It predicts the number of passengers on a bus when the bus arrives at the next couple bus stops along its route, allowing users to trade off travel time with comfort. The prediction is based on statistical models for boarding/alighting passengers at each bus stop, collected via two live real-time data sources: 1) the number of passengers on the bus, and 2) the number of passengers waiting at each bus stop. The former is collected through on-vehicle infrared sensors. The latter is collected through cameras installed on the ceiling of each bus stop. The camera periodically takes pictures of the bus stop, and then sends the images to a smartphone nearby (S_1 in Fig. 2). The smartphones at the bus stop, connected via ad-hoc WiFi, form a compute cluster. A DSPS, running on this cluster, counts the number of passengers in the images using the HaarTraining face detection algorithm [17]. Based on the live data and the statistical models, the DSPS at each bus stop predicts the number of passengers who will stay on, board and alight the bus when it arrives at this bus stop. The prediction results are also sent to the next bus stop along the route via the cellular network, and used for predictions at the next bus stop. The DSPS at each bus stop has the same structure, as shown in Fig. 2. The camera data is fed into S_1 and each tuple contains an image. The data from the previous bus stop is fed into S_0 and each tuple contains a predicted number of on-bus passengers when the bus leaves the previous bus stop.

The second application is SignalGuru [18], a smartphone application that predicts the transition time of a traffic light at an intersection and advises drivers on the optimal speed they should maintain so that the light is green when the drivers arrive at the intersection. In this way, cars can



S₀: data from previous intersection, **S₁**: data from smartphone cameras, **C**: color filter, **A**: shape filter, **M**: motion filter, **V**: voting filter, **G**: group, **P**: SVM prediction, **K**: sink (to next intersection).

Figure 3. The DSPS at each intersection of the SignalGuru application. The operators with the same color are on the same node.

cruise through the intersections without stopping, decreasing their fuel consumption significantly. SignalGuru leverages windshield-mounted smartphone to snap pictures of an intersection when the phone nears an intersection. It then shares the images with phones nearby via ad-hoc WiFi, and collaboratively learns the signal transition patterns. The kernel of SignalGuru is the image processing algorithm that detects a traffic signal in an image through color (red, yellow or green) filtering, shape (circle or arrow) filtering and motion filtering (traffic lights are always fixed by the roadside). After that, a Support Vector Machine (SVM) is used to train and predict the transition pattern. A DSPS, running on the smartphones in the vehicles at an intersection, completes the above steps. The prediction results are sent to the next intersection along the road, so drivers can know traffic signal transition times in advance. The DSPS running at each intersection is illustrated by Fig. 3. The camera data is fed into S_1 and each tuple contains an image. The data from previous intersection is fed into S_0 and each tuple contains the predicted traffic signal transition time. The time will be broadcast to all other phones nearby.

The characteristics of these two applications can be summarized as follows. The computations are performed at multiple regions (bus stops or intersections). A DSPS runs on smartphones in each region. Multiple DSPSs at different regions are cascaded, i.e. the results generated by the DSPS in one region are passed to the DSPS in the next region, allowing the applications to cover a large area. We believe that the region-based computation and the computation cascading are common requirements for many data analytics applications, such as water quality monitoring, road traffic management, etc., because today's applications rely increasingly on live data, and the data from multiple sources. The structure of these two applications motivates the design of MobiStreams.

III. MobiStreams Design

MobiStreams has a two-level architecture, as shown in Fig. 4. At the high level, a MobiStreams system is represented by a directed acyclic graph. Each node in the graph denotes a region. An arrow in the graph denotes the network connection between two regions and the direction of the data flow. In Fig. 4, region 1 is the upstream region of region 2,

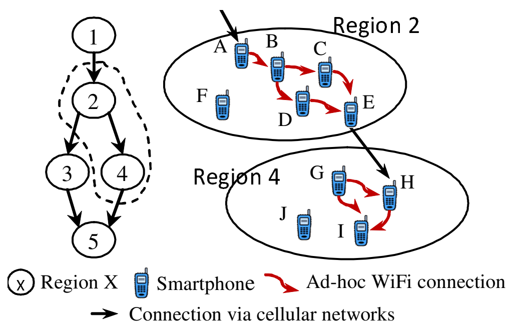


Figure 4. Two-level architecture of MobiStreams.

while regions 3 and 4 are region 2's downstream regions. A region is a small area within which smartphones are able to communicate with each other via ad-hoc WiFi. Today's ad-hoc WiFi range varies from 20∼100m. Due to the distance limitation, a region is usually a circular area with a diameter less than 20 meters (outdoor region can be larger). A bus stop, coffee shop or small auditorium is a good example of a region in real life. The distance between two regions would be much larger than the range of WiFi signals. So inter-region communication is through cellular networks, which can transport data over a long distance.

In MobiStreams, a computation task, specified by application developers, is assigned to each region. The task is computed on the phones in the region. Whenever the phones in a region receive input data from upstream neighbor regions or gather input data from the environment (e.g. taking pictures with their cameras), they run the computation task to process the data before sending result tuples to downstream neighbor regions. For example, in Fig. 4, the results of region 1 are sent to region 2, and the results of region 2 are sent to regions 3 and 4. Region 2 has multiple downstream neighbor regions and can send its results selectively. In other words, the data sent to regions 3 and 4 can differ.

The low level architecture of MobiStreams concerns the micro-structure of each region. The phones in a region, connected via ad-hoc WiFi, can be viewed as a cluster, where each phone is a node within the cluster. A DSPS runs on the cluster of each region. For example, in region 2 of Fig. 4, node A is a source node. It receives input data from region 1. Nodes B, C and D are computing nodes. Each of them computes some of the operations of the computation task assigned to region 2. Node E is a sink node, which sends result tuples to region 4. Node F is an idle node that is currently not computing the task. In region 4, both nodes G and H are source nodes. Node G gathers input data from the environment and node H receives input data from region 2. The data sent from G to H are not deemed input data since they are generated by a node inside the same region.

Besides the regions and phones, a MobiStreams system requires a controller – a global server node that can connect to all the phones in the regions via the cellular network. The controller is lightweight – it is used only for control purposes and is not involved in any data transmission between phones. In our applications, the communication between phones and the controller requires less than 2KB/s bandwidth, so the communication is not a problem. Furthermore, as pointed out in [10], the controller need not necessarily be a single point of failure, as hot standby architectures [19] and active standby techniques [20] can provide redundancy for the controller. Consequently, the controller is deemed reliable.

A. System Startup

A MobiStreams system starts up as follows. When a smartphone moves into a pre-defined region (detected by GPS) and has remained in the region for a period of time (defined by application developers), the phone registers itself with the controller via the cellular network. For those regions that contain sufficient phones (the threshold is determined by application developers), the controller assigns a pre-specified computation task to each of them: the controller splits up the task of a region into operators, transfers the code of each sub-task (or operator(s)) to a registered phone in the region, and connects the phones via ad-hoc WiFi. Thereafter, the operators start execution and phones are now ready to process input data. For sink nodes in a region, the controller instructs them to connect to the source nodes in their downstream neighbor regions via the cellular network. If a region does not have sufficient phones, the controller skips it and connects its upstream and downstream neighbor regions via the cellular network. This region will be started in the future when it has sufficient phones. Once the intra- and inter-region connections have been established, the MobiStreams system starts running. Since the DSPSs in different regions can boot independently in parallel, an application's boot time does not increase significantly when the region number increases. For BCP and SignalGuru with 4 regions, it takes about 1 minute to start.

B. Checkpointing Consistently on Faulty Smartphones

At runtime, MobiStreams periodically checkpoints the state of the DSPS in every region. MobiStreams uses a novel checkpointing approach that leverages tokens to coordinate amongst the multiple smartphones so a consistent checkpoint can be created across the entire DSPS at low overhead.

The checkpoints are performed independently at each region. So, we only discuss checkpointing in one region, which proceeds in three steps. First, the controller sends a notification to the source nodes in the region. This triggers every source node to send a token to each of its downstream neighbor nodes. A token is a piece of data embedded in the dataflow as an additional field in a tuple, and incurs very small overhead, e.g. less than 1% of tuple size in our two driving applications. Second, every computing node checkpoints its state once it receives the tokens from all

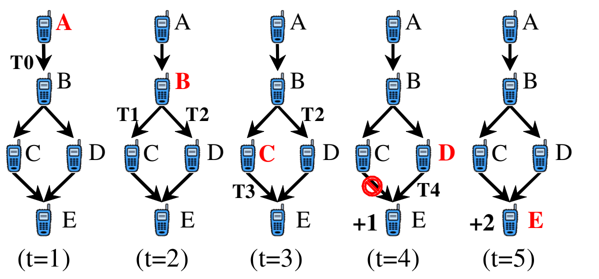


Figure 5. Execution snapshots illustrating token propagation and each node's checkpoint in a region. For clarity, the tuples preceding and succeeding the token in each stream are not shown.

upstream neighbor nodes. After that, a node forwards the token to each of its downstream neighbor nodes. In this way, the tokens are propagated along the DSPS, from source to sink operators. When all nodes in the DSPS have finished their own checkpoints, i.e. when the sink nodes checkpointed their state and percolated the tokens back to the controller, the checkpoint of this region is completed (a region's checkpoint contains the checkpoints of all computing nodes in the region). Third, every source node preserves all the input data since the Most Recent Checkpoint (MRC). This is *source preservation*. The input data and the checkpoint data will be kept until the next checkpoint of the region is completed. The data is saved on every node in the region (all source, sink, computing and idle nodes). This may seem like overkill, but is critical for a smartphone DSPS platform where all phones may leave the DSPS in between checkpoints (e.g. due to battery shortage, etc.).

Fig. 5 illustrates how a checkpoint is performed in a region that contains 5 smartphones. At time instant 1, the source node receives the notification from the controller. It then sends token T_0 to its downstream neighbor node. At time instant 2, node B receives the token, checkpoints its state and forwards tokens T_1 and T_2 to nodes C and D. Checkpointing is done asynchronously, i.e. the node spawns a separate thread for checkpointing, so as to minimize overhead. At time instant 3, node C also completes its checkpoint and thus forwards the token downstream. As node D runs more slowly than node C, token T_2 has not yet been processed. At time instant 4, node D receives the token T_2, checkpoints its state and forwards the token down. Node E receives one token T_3 from node C. Since node E has two upstream neighbor nodes, it cannot start its checkpoint yet. Node E thus stops processing tuples from node C, which guarantees that the state of node E is not corrupted by any tuple succeeding the token. Node E can still process tuples from node D since node E has not received any token from node D. At time instant 5, node E receives tokens from both upstream neighbor nodes and can then proceed onto checkpointing its state. After node E finishes its checkpoint, the checkpoint of this region is

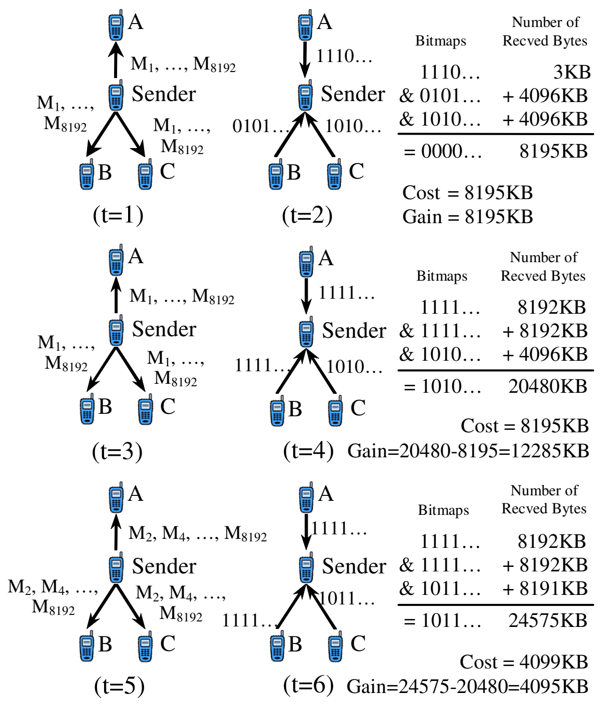


Figure 6. UDP broadcasting in a region containing 4 smartphones.

completed. With the help of tokens, no tuple will be saved twice or missed during checkpointing, ensuring consistent checkpointing at low overhead.

C. Checkpointing over Unreliable Wireless Networks

During checkpointing, a node's state has to be saved on other nodes, along with the input data of a region, to ensure the checkpoint persists regardless of node failures. Unlike server-based DSPS where checkpointing occurs over reliable and fast Ethernet, here, checkpoints have to be saved via unreliable, slow, low-bandwidth ad-hoc WiFi. MobiStreams solves this problem through multi-phase broadcasting leveraging both unreliable and reliable transport.

The first few phases utilize UDP. A node's checkpoint data is partitioned into a series of 1KB blocks (the last block may be less than 1KB). Each UDP message sent by the node contains one data block and a block description that comprises the id of the operator running on the node, the version of this checkpoint and the sequence number of this block. The block description is tiny, so message size remains around 1KB. The small message size is adopted as large UDP messages are more susceptible to a lossy network due to message fragmentation. On the receivers' end, every node records the received messages. According to UDP's datagram property, UDP preserves the boundaries between messages and a message will be dropped completely as long as a part of the message has not been received. After all messages have been broadcast, the sender queries the receiving nodes on which messages have been received. Each receiver returns a bitmap, where each bit indicates if a corresponding message is lost (0) or received (1). The number of bits in the bitmap thus equals the number of messages broadcast by the sender. Based on the bitmaps from all receivers, the sender can then infer the lost messages (message that is not received by at least one receiver) during the broadcast. The sender then broadcasts the *lost* messages again, and repeats this process of broadcasting and querying, until cost exceeds gain, both defined in terms of the number of bytes sent or received.

Fig. 6 dives into the details of MobiStreams' multi-phase broadcasting through a walk-through example. There are four nodes in the region. The size of the checkpoint data on the sending node is 8MB (8192 messages). At time instant 1, the sender broadcasts all the messages, $M_1, ..., M_{8192}$, to nodes A, B and C. At time instant 2, the sender completed the broadcasting and querying of receivers. Node A has unfortunately received only the first 3 messages, so it returns a bitmap with only three 1s at the head. Node B has received all even messages, thus returning a bitmap with alternate 0s and 1s. Node C has received all odd messages, so it returns a bitmap with alternate 1s and 0s. The size of each bitmap is 1KB. With these bitmaps, the sender does some calculations. 1) The sender ANDs all bitmaps and the result is a bitmap with only 0s, in other words, every message has at least one receiver who failed to receive it. 2) The sender counts the number of bytes received by receivers. The numbers of received bytes on nodes A, B and C are 3KB, 4096KB and 4096KB. Therefore, the total number is 8195KB. 3) The sender evaluates the gain of the broadcast, calculates as the number of received bytes before and after this broadcast. Before the broadcast, the number of received bytes is zero. Therefore, the gain of the broadcast is the 8195KB received bytes. 4) The sender computes the cost of the broadcast. It has sent 8192 messages, and received 3 bitmap messages. The total cost is 8195KB transferred over the network. Since the cost (8195KB) is not larger than the gain (8195KB), the sender starts the second broadcast at time instant 3.

According to the ANDed bitmap at time instant 2, every message has to be resent. The sender then broadcasts all the messages again. At time instant 4, the sender completed the second broadcast and obtained the returned bitmaps. Node A and B have received all the messages, so they return bitmaps full of 1s. Node C has not received any message in the second broadcast. Its returned bitmap remains unchanged from the earlier phase. Based on these bitmaps, the sender performs the same calculations. The ANDed bitmap is 1010.., which means all the even messages have to be resent. The total number of received bytes is 20480KB. The gain is the 12285KB newly received in the second broadcast and the cost is the 8195KB sent over the network. Since the cost (8195KB) is not larger than the gain (12285KB), the sender starts the third broadcast at time instant 5. According to the ANDed bitmap at time instant 4, only even messages, $M_2, M_4, ..., M_{8192}$, are resent. At time instant 6, node C returns a bitmap, indicating it has received all messages except M_2. According to the same algorithm, the gain of the third broadcast is 4095KB while the cost is 4099KB. Hence, the sender finally terminates the UDP broadcasting.

UDP broadcasting cannot guarantee that every node receives the entire checkpoint data. Therefore, the final phase over reliable TCP is introduced. In this phase, all the nodes in a region are organized as a tree. The sender is also a node in the tree. Data are sent from the sender to the tree root first, and then flows from the root to the leaves. The tree structure is created by the controller and changes only when a phone fails, enters or leaves the region.

D. Failure Detection and Recovery

A smartphone can fail without any prior indication. Failure detection in MobiStreams relies on the controller: the controller periodically pings the source nodes in each region via the cellular network. A source node is deemed to have failed if it does not respond within a timeout period. The computing and sink nodes in each region are monitored by their upstream neighbor nodes. If a node detects a failure of its downstream neighbor nodes, it reports to the controller immediately via the cellular network. Besides the failure of its downstream neighbor nodes, a node can also actively report its own failure to the controller, for example, when its battery is at chronic levels.

If one or more computing nodes fail in a region, the controller selects other nodes in the same region to replace the failed nodes. Since every node in the region has a copy of the MRC data and the input data since MRC, the controller can select any healthy node in the region (idle nodes are preferred). The controller sends the code that was executed by the failed nodes to the newly selected nodes and restarts the code execution on those nodes. Then the controller rebuilds the WiFi connections between the nodes and asks all computing nodes, including the newly selected nodes, to reload the operator state from the MRC data. In such a manner, the states of the computing nodes are restored to MRC. This is classic checkpoint restoration. After restoration, all the input data received after MRC needs to be processed again. The source nodes replay the input data and all the computing nodes reprocess the data, at which point, all the computing nodes have rebuilt their states to the point precisely before the failure. This procedure is termed catch-up. Sink nodes discard all results generated during catch-up, so as not to pollute other regions. After catch-up, the DSPS of this region is recovered and can process tuples as normal. All failed nodes will be unregistered with the controller. If there are no sufficient healthy nodes in a region after some nodes fail (e.g. all nodes fail), the controller stops the computation task of this region and connects this regions' upstream and downstream neighbors

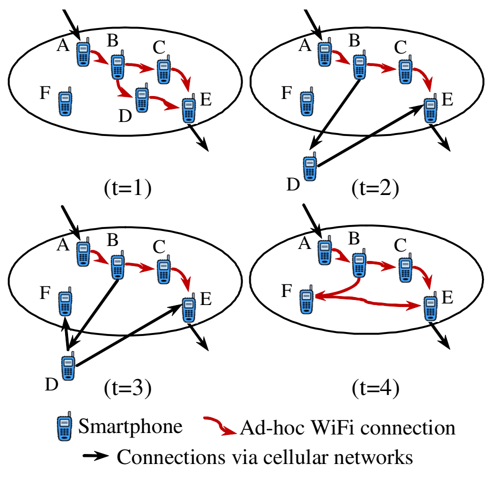


Figure 7. Actions triggered upon a smartphone's departure. 1) Before departure. 2) Urgent mode. 3) State transfer. 4) Node replacement.

(bypassing this region). This region can be restarted when there are sufficient nodes in it. If failures happen during a checkpoint is being performed, the DSPS can be still recovered as above, just ignoring the partial checkpoint data that have been saved so far.

Source and sink nodes can also fail. However, it is easier to recover them since they are stateless (source and sink operators are only used to maintain inter-region connections). Once a source or sink node fails, the controller selects another healthy node to replace the failed node, and instructs the new node to rebuild the inter-region connections with upstream or downstream neighbor regions.

The restoration in MobiStreams scales to many nodes because nodes are restarted in parallel and each node reads the state data from local storage. Restoration of individual nodes thus occurs simultaneously. Catch-up time varies with the checkpoint period. The longer the period, the more input tuples are generally saved by source preservation. The source nodes then require a longer time to replay the tuples. However, the catch-up time should be no more than a checkpoint period.

E. Mobility

So far, we assume that smartphones are static in regions, i.e. a phone will not leave once it enters a region. Fig. 7 shows the scenario when a computing node (node D) leaves a region. When node D is out of the region, the distances between nodes D, B and E become large, and then the WiFi connections between them are broken. These nodes then switch to an urgent mode, in which they use the cellular network to transmit tuples, as shown at time instant 2 in Fig. 7. Nodes B, D and E report the urgent mode to the controller. According to the GPS on node D, the controller knows that node D is leaving. The controller then selects another node in the region to replace node D. In this case, the controller selects node F and asks node D to transfer its state to node F via the cellular network at time instant 3. Once node F has the same state as node D, the controller instructs nodes B, F and E to rebuild WiFi connections at time instant 4. After that, the DSPS of this region returns to normal mode and continues tuple processing. The node that has left a region will be unregistered with the controller. Here, a special case needs extra explanation. At time instant 2, if node D's GPS reports that node D is still in the region, there are two possibilities: 1) the WiFi connections are broken because of disturbances, rather than a node departure, or 2) the GPS is not accurate. The controller thus asks nodes D, B and E to tentatively rebuild the WiFi connections. If they cannot succeed after several attempts, node D is treated as if it has actually left the region.

If multiple nodes leave a region simultaneously, the structure of the DSPS can still be kept in urgent mode, but some network connections between nodes will rely on the cellular network. This region's DSPS can continue working, although the performance will be degraded until the controller replaces the leaving nodes. If there are no sufficient nodes remaining in the region after some nodes leave, the controller stops the computation task of this region and connects this region's upstream and downstream neighbors (bypassing this region). If a source or sink node leaves a region, the controller selects another node in the region to replace it. Since source and sink nodes are stateless, it is not necessary to transfer state from the leaving node to the newly selected node. However, the controller has to instruct the newly selected node to rebuild the inter-region connection accordingly. If an idle node leaves (detected by GPS), it just unregisters itself with the controller and deletes all checkpoint data saved on it.

IV. Evaluation

We implemented MobiStreams as a middleware running atop iOS, along with the two driving applications as DSPS programs. MobiStreams comprises about 60,000 lines of code, while the BCP and SignalGuru DSPS consist of 5000 and 4000 lines of code respectively. Each application contains 4 regions in all experiments. The regions are cascaded in a line. Each region has 8 iPhone 3GSs (600MHz Cortex-A8 CPU, 256MB RAM and 16GB storage). The measured bandwidth of the ad-hoc WiFi network in each region is 1$\sim$5Mbps. The 3G uplink and downlink bandwidths are 0.016$\sim$0.32Mbps and 0.35$\sim$1.14Mbps respectively. The 3G cellular network is used for inter-region communication. The controller pings the source nodes every 30 seconds and the timeout period is 10 seconds. The checkpoint period in MobiStreams is 5 minutes. All results are averaged across 5 runs. To measure latency, we record in each tuple the times when it enters and leaves the system, and average the duration across all the tuples in a time window. To measure

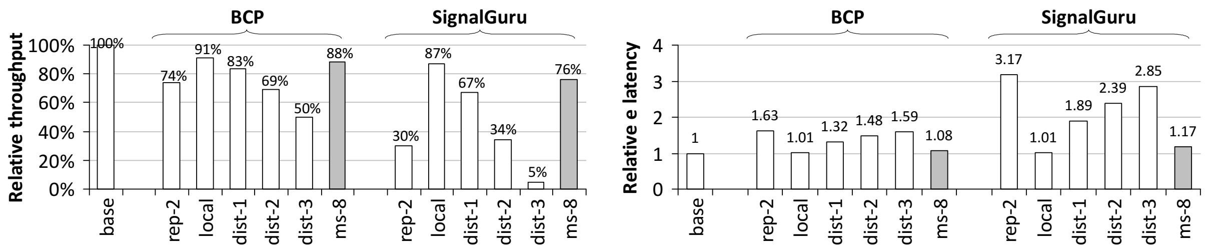


Figure 8. Relative throughput and latency of BCP and SignalGuru on a smartphone platform with different fault tolerance schemes. Values are normalized to the result of the baseline non-fault-tolerant system.

Table I
MOBISTREAMS VS SERVER-BASED DSPS.

	Per-region throughput (tuples/second)		Latency (seconds)	
	BCP	SignalGuru	BCP	SignalGuru
Server-based DSPS	0.011∼0.22	0.018∼0.36	60∼750	40∼540
MobiStreams[1]	0.54	0.8	32	25
MobiStreams[2]	0.52	0.74	36	30
MobiStreams[3]	0.48	0.64	39	36

1 Fault tolerance function is turned off. 2 Fault tolerance function is turned on and a phone leaves its region every five minutes. 3 Fault tolerance function is turned on and a phone fails every five minutes.

throughput, we count the number of output tuples per second when the system is steady.

A. MobiStreams vs Server-based DSPS

We first compare MobiStreams (Fig. 1.d) with the traditional server-based DSPS (Fig. 1.c). Table I shows the results. The server-based DSPS is hindered by the low bandwidth of the uplink cellular network. The fault tolerance function has no impact on overall performance. Even if it is turned off, the performance of the server-based DSPS is still low. In Table I, a phone failure means that a phone stops working and MobiStreams has to recover the entire DSPS of the region, incurring the overhead of restoration and catch-up. A phone departure indicates that a phone leaves a region and the state of the leaving phone has to be transferred to another phone in the region. Phone departures incur no overhead of restoration and catch-up. In all these scenarios, MobiStreams beats the server-based DSPS.

B. MobiStreams vs Prior State-of-the-art

In the past decade, two kinds of fault tolerance schemes have been proposed for DSPSs running on server clusters: replication-based [9–11] and checkpoint-based [9, 12–15] schemes. In replication-based schemes, a DSPS runs k+1 replicas of each operator to tolerate up to k simultaneous failures. When an operator fails, one of its replicas takes over its work immediately because the replica has maintained the same state as the failed operator. In checkpoint-based schemes, every node performs two key functions. 1) Every node periodically checkpoints (saves) operators' running state in local or remote storage. 2) Every operator retains its output tuples until these tuples have been checkpointed by the downstream operators. This is called input preservation [9]. When an operator fails, the operator is restarted from its MRC. Its upstream operators then resend all the tuples that the failed operator had processed since its MRC. The restarted operator rebuilds the same state as the failed operator after it processes all these tuples.

Therefore, we define and compare the following fault tolerance schemes. The purpose is to show that MobiStreams' scheme is better than the existing ones. 1) The **baseline** system (base) without fault tolerance. 2) **Active standby** (rep-2): A replication-based scheme that runs two replicas for each operator. It can tolerate only single-node failures. 3) **Local checkpointing** (local): A checkpoint-based scheme that saves operators' state to the local storage of each node. This scheme assumes that each node can be restarted after a failure and the data in its storage will not be lost after the restart. It is not a realistic fault model in the context of smartphones, but represents an upper bound in performance for fault-tolerance schemes and is thus useful as a benchmark. 4) **Distributed checkpointing** (dist-n): A checkpoint-based scheme that saves operators' state to n other nodes. It can tolerate n-node failures. This is the only scheme here that provides decent fault tolerance, with dist-3 tolerating up to 3 simultaneous node failures.

We compare MobiStreams with prior state-of-the-art on a smartphone platform in two scenarios: without and with faults. In the first scenario, phones do not leave a region or fail. However, the systems still have to pay the overhead of source/input preservation and checkpointing/replication. There is just no need for failure recovery, which will be evaluated in the second scenario. Fig. 8 shows the results. In the figure, MobiStreams is labeled "ms-n", where n denotes the number of nodes (phones) in a region. It can be seen that local scheme exhibits the best performance. However, it does not provide realistic fault tolerance. It serves as a benchmark for the lowest achievable performance overhead. We see that MobiStreams' performance overhead is close to local. Compared to rep-2 and dist-n, MobiStreams offers better performance: 230% throughput increase and 40% latency decrease, on average, in BCP and SignalGuru.

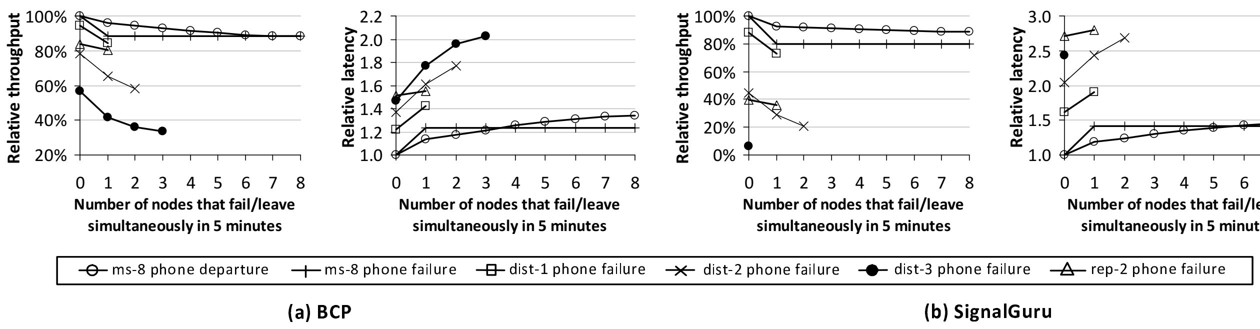


Figure 9. Relative throughput and latency of BCP and SignalGuru when a n-node failure or departure occurs within one checkpoint period.

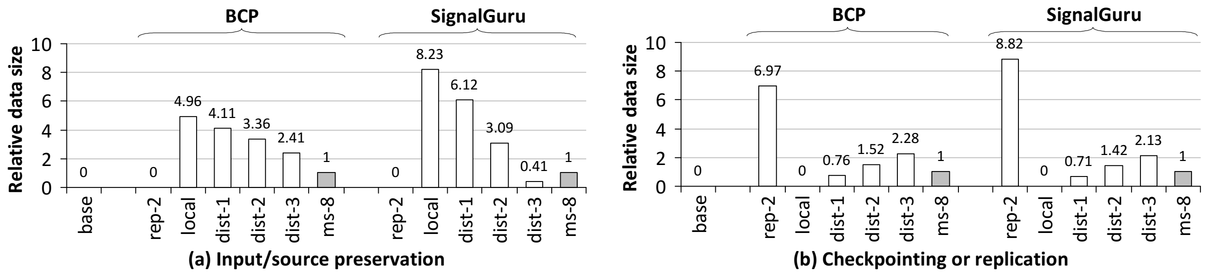


Figure 10. Relative size of the data saved due to input/source preservation and the data sent over the network due to checkpointing/replication. Values are normalized to the data size of MobiStreams.

In the second scenario, we introduce phone failures and departures. So we can evaluate the recovery overhead. Fig. 9 shows the applications' performance when n node(s) fail or depart within one checkpoint period. The latency and throughput in Fig. 9 include the applications' down time and recovery time. We have three findings in the figure. First, no matter how many nodes fail, MobiStreams always recovers all the nodes in the DSPS. So the failure recovery overhead is constant when the number of failed nodes increases. It can be seen that MobiStreams' performance curve (ms-8 phone failure) is a horizontal line. Second, dist-n's performance curve has only $n+1$ points because it can only handle up to n-node failures. rep-2's curve contains only two points because it can only handle single-node failure. Both rep-2 and dist-n have much poorer performance than MobiStreams, and dist-n's performance deteriorates as n increases. In SignalGuru, dist-3's curve contains only one point as it was unable to recover the system within 5 minutes. Last, MobiStreams' node departure overhead is less than the failure recovery overhead of rep-2 and dist-n, and less than the failure recovery overhead of MobiStreams in most cases. This is reasonable. Node failures are much more involved than node departures as failures trigger recovery and catch-up while departures require just a state transfer. However, departure overhead can exceed failure recovery overhead in MobiStreams when the number of phones leaving simultaneously is large (the second subgraph in Fig. 9). We believe the reason is that the state transfer involves cellular network communication, and when many phones are using the cellular network at the same time, the network limits the overall performance. The node departure experiment is only conducted on MobiStreams because prior fault tolerance schemes cannot handle node departures (they are designed for servers).

V. Related Work

Prior DSPS. There have been substantial prior art tackling fault tolerance in the field of DSPS. They can be mainly classified as replication-based schemes [9–11] and checkpoint-based schemes [9, 12–15]. Flux [10] and Borealis DPC [11] are replication-based schemes. The difference is that Borealis DPC allows itself to produce inaccurate but timely outputs based on partial inputs when some failed nodes cannot be recovered. Upstream backup [9], where every node acts as a backup for its downstream neighbors, can reduce the overhead of replication. However, upstream backup cannot effectively support operators with large windows, and it only handles single node failure. Several replication-based schemes were compared in [9], with the authors concluding that each scheme covers a complementary portion of the solution space. Checkpoint-based schemes adopt periodical checkpointing and input preservation for fault tolerance. Passive standby [9] saves the running state in memory, avoiding disk I/O but limiting state size. LSS [14] sacrifices data consistency for performance, dropping tuples from a full buffer instead of saving them into disk. Meteor Shower

[15] handles many faults but saves checkpoint data in a single, centralized storage server. Cooperative HA Solution [12] saves each HAU's state on other computing nodes in the DSPS, thus avoiding a central storage system. SGuard [13] adopts asynchronous checkpointing and distributed checkpointing (scattering the checkpoint state onto multiple storage nodes). All these schemes are targeted at server-based DSPSs and not suitable for phone-based DSPSs, as they cannot handle many faults and/or can lead to high performance overhead. In this paper, rep-2 is representative of Flux and Borealis, while dist-n is modeled after Cooperative HA solution and SGuard.

Fault Tolerance in Databases and Distributed Computing. Reliability in databases and distributed computing has been extensively investigated in the past, and offers inspiration for MobiStreams. For instance, there has been extensive prior work in checkpointing algorithms for traditional databases. However, classic log-based algorithms, such as ARIES or fuzzy checkpointing [21, 22], exhibit unacceptable overhead for applications with very frequent updates, such as DSPS [23]. In the area of distributed computing, sophisticated checkpointing methods, such as virtual machines [24, 25], have been explored. However, if used for stream applications, these heavyweight methods can lead to significantly worse performance than the stream-specific methods discussed in this paper. For example, virtual machines incur 10X latency in stream applications in comparison with SGuard [13]. Besides, the concepts of saving data at sources and propagating tokens in a network have been studied in reliable multicast algorithms [26] and Chandy-Lamport algorithm for consistent snapshots [27]. Leveraging them in a DSPS, like MobiStreams' token in Section III-B, is unique though.

References

[1] MediaTek Company. The Power of 8 MediaTek True Octa-Core Solution. http://www.mediatek.com/_en/Event/201307_TrueOctaCore/tureOcta.php.

[2] Nielsen Company. Mobile Majority: U.S. Smartphone Ownership Tops 60%. http://www.nielsen.com/us/en/newswire/2013/mobile-majority–u-s–smartphone-ownership-tops-60-.html.

[3] Singapore Leads the World on Smartphone Penetration. http://wallblog.co.uk/2013/01/11/singapore-leads-the-world-on-smartphone-penetration-bad-news-for-apple.

[4] Cisco company. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011-2016.

[5] IBM. InfoSphere Streams Product. http://www-01.ibm.com/software/data/infosphere/streams.

[6] Yahoo. S4 Project. http://incubator.apache.org/s4.

[7] StreamBase. http://www.streambase.com.

[8] IBM. IBM Ushers In Era Of Stream Computing. http://www-03.ibm.com/press/us/en/pressrelease/27508.wss.

[9] J.-H. Hwang, et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*. IEEE, 2005.

[10] M.A. Shah, et al. Highly Available, Fault-Tolerant, Parallel Dataflows. In *SIGMOD*. ACM, 2004.

[11] M. Balazinska, et al. Fault-Tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. on Database Systems*, 33(1), 2008.

[12] J.-H Hwang, et al. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE*. IEEE, 2007.

[13] Y.C. Kwon, et al. Fault-tolerant Stream Processing Using A Distributed, Replicated File System. *PVLDB*, 2008.

[14] Q. Zhu, et al. Supporting Fault-Tolerance in Streaming Grid Applications. In *IPDPS*. IEEE, 2008.

[15] H. Wang, et al. Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers. In *IPDPS*, pages 1180–1191, 2012.

[16] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Kaufmann, 1993.

[17] R. Lienhart, et al. Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection. In *DAGM 25th Pattern Recognition Symposium*, 2003.

[18] E. Koukoumidis, et al. SignalGuru: Leveraging Mobile Phones for Collaborative Traffic Signal Schedule Advisory. In *MobiSys*, 2011.

[19] S.-O. Hvasshovd, et al. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *VLDB*, 1995.

[20] E. Lau and S. Madden. An Integrated Approach to Recovery and High Availability in An Updatable, Distributed Data Warehouse. In *VLDB*, 2006.

[21] C. Mohan, et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *TODS*. ACM, 1992.

[22] K. Salem and H. Garcia-Molina. Checkpointing Memory-Resident Database. In *ICDE*, 1989.

[23] T. Cao, et al. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *VLDB*, 2011.

[24] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. In *ACM Symposium on Operating Systems Principles*, 1995.

[25] J. Zhu, et al. Improving the Performance of Hypervisor-Based Fault Tolerance. In *IPDPS*, 2010.

[26] C. Diot, et al. Multipoint Communication: A Survey of Protocols, Functions, and Mechanisms. In *IEEE Journal on Selected Areas in Communications*, 1997.

[27] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *ACM Trans. on Computer Systems*, 1985.