Concentus: Applying Stream Processing to Online Collective Interaction

Adam Roughton, Ian Warren, Beryl Plimmer
Department of Computer Science
The University of Auckland
Auckland, New Zealand
Email: {arou012, ian-w, beryl}@cs.auckland.ac.nz

Abstract—The collective experience is the experience of unity, belonging, and purpose that occurs when large numbers of people come together and perceive themselves and others as part of a single social entity; and interact with each another accordingly. We are exploring how the collective experience can be supported in a fully computer-mediated environment through activities where a virtual crowd performs synchronous collective action over a shared focal state (e.g. collectively controlling a character in a game; pulsing text-based messages in time to form collective chants). Supporting collective interaction requires a system architecture that is able to process large numbers of input actions into an aggregated collective representation at low latency. We have created a scalable distributed system called Concentus that applies approaches found in distributed stream processing to online collective interaction. Concentus allows for different implementations of aggregation engine, the primary component of the system, to be measured in-situ with other core components (e.g. client connection handlers). We have evaluated the performance of two aggregation approaches: one based on Spark Streaming, a general purpose distributed stream processing engine, and another that performs aggregation on a single thread on one machine; and have measured their performance against the key metric of interaction latency (time from input submission to perceiving the effect on the shared state) as the crowd size scales. The evaluation revealed that both approaches are capable of supporting 50,000 participants with latencies under 1 second; with the single threaded approach performing better on smaller data sizes, and Spark Streaming on larger data sets. We discuss the implications on collective application design.

I. INTRODUCTION

Online social environments of scale have become the norm in recent years, with the likes of Facebook and Twitter dominating social networking, and massively multiplayer online games (MMOG) bringing large numbers of people together in vast virtual worlds. Although these environments allow large numbers of people to interact, sometimes towards collective objectives [1]; the collective experience remains unexplored online. The collective experience is the experience of unity or one-ness that occurs when large numbers of socially co-present people engage together as a larger social entity. Common examples include cheering in a stadium crowd with other supporters; collectively chanting at a protest rally; singing together as part of a choir; or even in smaller moments such as in an audience applauding a performer. In these moments people feel a sense of intimacy with one another, a sense of belonging, and validation of their perspective of the world (even if just momentarily) as others are seen to share this in common [2].

Collective applications in the crowd-computer interaction domain [3]–[6] have demonstrated the effective application of technology in supporting collective experiences in physical audiences and crowds by providing novel opportunities for engaging in shared action. A prominent example is crowd pong [5], [6], an activity that requires an audience to coordinate their input (captured using machine vision techniques) to collectively control a pong paddle presented on a large display. Collective applications create a context where interacting together is meaningful and the collective effort is easily perceived. The social processes that drive the collective experience also operate in fully computer-mediated environments (online), perhaps to an even greater degree than physically collocated crowds [7], [8]. This, along with the prospect of any-time any-where collective engagement, motivates the exploration of online collective applications.

We have developed an application model for online collective applications, which we motivate with the following example application concepts: an application where a virtual crowd expresses itself collectively by ‘chanting’ small phrases, words, and emoticons while watching a live event; and a reincarnation of crowd pong in a fully virtual environment. Unlike traditional distributed interactive applications, where the attention of users is divergent [9]; the core interaction proposed for online collective applications is interaction focussed on one shared view, reflecting the real-time aggregation of individual actions into a collective product.

To support the proposed interaction, a system architecture is needed that supports both the low latency aggregation of a high throughput of input into a collective state; and components for broadcasting updates, processing input events, and handling a large number of concurrently connected users. Stream processing approaches have been applied to domains with similar requirements such as stock exchanges and real-time analytics, where a large number of inputs are processed against shared state [10]–[13]. Applying approaches from the stream

1Social presence is the sense that others are temporally present (i.e. actively interacting) for the purposes of communication [2].

2Pong is a game that would otherwise be played by a single player.
processing domain, we have developed a system architecture named Concentus to support online collective applications. Concentus is a Latin noun that translates to: “singing together, harmony, melody; concord” [14].

The contributions of this paper are as follows:

- An application model for online collective applications for supporting the collective experience online (§IV).
- An implementation of a framework for supporting online collective applications, allowing various approaches for real-time aggregation to be compared for performance in-situ ($§V$).
- An evaluation of the performance of two candidate approaches as they perform with three conceptual applications that strain the aggregation execution in different ways ($§VI$).
- Implications of the evaluation on online collective application design ($§VII$).

II. Motivation

This section motivates supporting the collective experience, presents two conceptual applications we will be using throughout the remainder of the paper, and finally derives architectural requirements for online collective applications that we target in Concentus.

A. Collective Experience

Experiencing an event together with a crowd of people perceived to share the same outlook and values has been shown to provide participants experiential outcomes of unity, belonging, validation, and shared joy; outcomes seldom found in everyday life [3]. The basis of these experiential outcomes is a shift in the way people perceive and interact with one another when immersed in an crowd sharing a collective purpose (e.g. a stadium crowd) [3].

Social psychology research provides strong evidence that our perception of others is framed by the current social context (the frame of reference for behaviour) [15]: in day-to-day life this context is typically of an interpersonal nature, where we interact with others as individuals (me versus you). In a collective context, such as a political rally or sports match, the social context involves interaction between groups or categories (teams, nationalities, people sharing a political position), rather than individuals. In this context, perception changes from seeing people as individuals, to seeing groups and crowds as single social entities (us versus them) [8], [15]. This shift in perception has important implications on the way we behave towards others in the context: when the social perception shifts to a collective level, those who we perceive to share an identity with us (e.g. supporting the same team, fighting for the same cause) are seen as more trustworthy and socially attractive (likeable) [16]. In a stadium crowd, people who are otherwise strangers can find themselves interacting as if they have been long-time friends, to the degree that people who have only just met have been known to hug one another with joy when their shared identity is empowered in some way (e.g. their team scoring a goal) [3]. It is this shift in social interaction that underpins the collective experience, and is the target of collective applications.

Engaging crowds in a shared activity is a powerful way of both creating a collective context, and allowing social interaction in that context to take place as it provides the following:

- **Shared Fate**: to perceive others as sharing a group identity with ourselves, the social context must make a collective level distinction socially meaningful [15]. Sharing a collective fate with others (we either rise or fall together) motivates the perception of shared identity as it drives people to compare themselves and others on some collective dimension (e.g. team score, overcoming a collective objective) [15].

- **Synchronous Collective Action**: although the social context might make a collective comparison meaningful, evidence that others also share this perception is needed before collective interaction can take place. Collective action is a powerful means of expressing collective identity as it signals to all those participating that everyone perceives the situation in a similar way. Additionally, acting in unison (or in synchrony) has been shown to increase the perception of shared identity by making the crowd appear more like a single social entity [17], [18].

B. Application Concepts

We have incorporated these properties into two conceptual online collective applications which we will use as running examples throughout the remainder of this paper: CrowdAloud! and Collective Pong. Each application is designed to encourage the collective to perform collective action in unison, and reach a consensus on the content of that action. Additionally each application targets a different type of collective interaction:

- CrowdAloud!: expression of shared identity and emotion; feedback and validation of shared membership and beliefs
- CollectivePong: acting in synchrony; coordinating to achieve a collective objective

1) CrowdAloud!: The intention of this application is to allow participants to express themselves collectively as they spectate live events such as sports fixtures, political debates, or television shows; by chanting phrases as a collective. Participants submit phrases (up to 140 characters in length) or symbols (e.g. emoticons) while they spectate. A shared collective canvas displays the top 25 phrases in a word cloud, with each phrase sized proportionally to the current percentage of collective support (the value score over the total possible score). Phrases are aggregated together by directly matching on the phrase content. The score for each individual phrase is dependent on the time since the phrase was first submitted ($t_d$). Each time a candidate value is generated from a phrase effect, the following score function is used to represent ephemeral
Lag is a decrease in the responsiveness of the application resulting from an increase in latency (caused by the network, processing, or a combination of the two).

\[ \text{score}(t_d) = \begin{cases} 
100 - \frac{10d}{5}, & t_d < 5 \\
90 - 90 \frac{t_d - 5}{5}, & 5 \leq t_d < 10 \\
0, & t_d \geq 10 
\end{cases} \]

To encourage coordination, phrases must meet a minimal level of support to become visible on the canvas (as well as meeting the top 25 requirement). The minimal score associated with each phrase must have a score equivalent to 10% of the collective chanting the phrase simultaneously (i.e. a \( \text{score} = 10 \frac{c}{10} \) where \( c \) is the total crowd size).

2) **Collective Pong**: Participants collectively control a pong paddle while playing against a computer controlled component. Each participant controls their own individual paddle representation that reflects their intended position for the collective paddle. Rather than taking the average position of individual paddles, the collective paddle is represented as a vertical bar (paddle bar) that presents the superimposition of individual paddles. For a position on the paddle bar to be impassable to the pong ball, some threshold number of participants (relative to the crowd size) must have their individual paddles overlapping it. In this way the collective is able to see visually the pattern of consensus that is being established around various positions. The rationale behind using superposition over averaging is to ensure intention is communicated: if the collective position was an average, a participant might shift their paddle position to an extreme position (relative to their intended position) to skew the average in a particular direction. This would make it difficult for neighbours to coordinate with them on collective action. Likewise if the collective paddle position was only the most popular position, this would make it hard for participants to reach consensus as there is difficulty in exactly matching two positions.

### III. RELATED WORK

Massively multiplayer online games (MMOG) have players interact in a fully synchronous world that is populated with other players. Though players share the same world, interaction in the game is for the most part individual or small-group based, with interactions involving large numbers of people tending to be the exception rather than the norm [9]. Where large scale interaction does occur, the numbers involved tend to be in the hundreds or early thousands: on the 27th of January 2013, the MMOG EVE Online had one of the biggest battles in the history of the game involving approximately 3000 people [19]. This non-typical concentration of players caused the system to suffer lag effects due to the system load. Strategies are employed by MMOG engines to ensure that the number of people sharing an interactive space is limited to that of server capacity. Common strategies employed to allow systems to scale include partitioning worlds such that players interact with a subset of the population on replicas, zoning parts of world to run on different machines [20], or employing wait queues on parts of the game world that have become temporarily overpopulated. Such strategies work against the model proposed for online crowd applications, where the main interaction space involves the input of all participants. One important difference between traditional MMOGs and the proposed model is that user actions in online crowd applications are not represented on the shared state individually, but rather their combined aggregation; presenting opportunities for optimisation.

ROAR is a conceptual social application proposed by [21] that allows distributed participants to chat with one another while spectating a live event. Participants are segregated into sections where they are able to directly chat with other participants much like Internet Relay Chat (IRC). As a point of difference from other second screen tools, it provides a shared crowd level visualisation that presents the top words occurring as people chat with one another. Words are bucketed into windows of 10 seconds and an aggregation algorithm is applied to determine the top word counts. Although this concept allows for a crowd level awareness, it differs in the nature of the interaction proposed by this paper. We are proposing applications where participants are required to coordinate to form a collective output. An important part of this is the synchronicity of the action: the knowledge that our on-going joint action is creating the observed effect. As such we require an approach that offers latencies on the order of milliseconds.

### IV. APPLICATION MODEL

To support the collective experience online, we propose a class of application with the following objectives:
1. Frame the interaction with the application through a collective lens by requiring that participants respond to a shared collective objective (shared fate) with a unified collective response (synchronous collective action). The application context should be such that acting collectively is the only meaningful interaction.

2. Connect participants in such a way that they are able to directly interact with some subset of the collective to build on one another’s collective action (to provide instrumental support by snowballing it to the rest of the crowd), and express emotional support (e.g. ‘Awesome!’).

We present the following application model (fig. 1) for this interaction:

**Shared State**: all interaction with the application occurs with reference to a common state shared by all participants; this state acts as a shared focal point for interaction. For example, this might be a shared canvas that participants collectively draw on, or a scene containing a character all participants collectively control.

**Collective Variables (CollectiveVariable)**: rather than directly mutating the shared state, participants have agency over collective variables which serve as the input to the shared state. Each variable represents what would otherwise be a directly controllable input into the shared state if only one participant was using the application (e.g. a player’s position in a game).

**Candidate Values (CandidateValue)**: the value of a collective variable is not scalar, but is rather a top \( N \) set of candidate values\(^3\) ranked by a score associated with each value. When participants perform actions, they indirectly specify (see Effect State) a value for the collective variables associated with the action. Each collective variable generated has both a data component for the value, a score representing the amount of support the value has, and a group key that specifies how the value should be combined with other values. To generate the value of a collective variable at time \( t \), the set of all candidate values for time \( t \) are generated, and then aggregated together as follows:

1. Candidate values are grouped by their value data (the matching strategy for this grouping is determined by the application implementation)
2. For each matched candidate value pair, a new candidate value is created with an aggregated value data (again determined by the application implementation), and a new score that is the sum of the input values

**Tick Timer**: Like other distributed interactive applications, the shared state is generated at a fixed frequency (tick), rather than generating a new state for each action. Given limited network and processor resources, this allows distributed clients to maintain synchrony with the shared state while allowing time for this state to be generated and transmitted.

**Effect State (Effect)**: when participants perform actions, their action is not immediately aggregated into the shared state, but rather creates an intermediate Effect state. Effects contain the timestamp when they were created, a type ID, and data from the processing of the source action. During the generation of a shared state update at time \( t \), all active Effects are used to generate CandidateValues for one or more CollectiveVariables using the given time. Each CandidateValue generated also has a score component that can be used to represent the strength of support the participant has for the value given the current time. At any given time, a single participant can only have one active Effect with the same effect type ID.

V. IMPLEMENTATION

The implementation of Concentus has two primary objectives:

1. Minimise the latency from a participant performing an action to the reception of an update containing the effects of that action by all participants (including the acting participant)
2. Support a large number of participants all performing actions over the shared state concurrently

For an implementation to meet these objectives, it must be capable of processing a high throughput of events at low latency. A standard approach to this problem is to introduce parallelism into the system: by executing parts of the pipeline in parallel, we can increase the overall throughput of the system and thus reduce the latency of actions that would otherwise be waiting in queues. Indeed we support parallelism with a distributed system architecture.

However, just as important as supporting parallelism for systems supporting collective interaction is also ensuring that the execution path through each component is maximally efficient. The first motivation for this is one of practicality: introducing more instances of processing components increases the number of servers needed by the system. As we wish to minimally support a large number of concurrent users (on the order of thousands to tens of thousands), the number of servers must be minimised to ensure the cost is non-prohibitive.

Secondly, the performance of individual component instances has important implications on latency regardless of the number of instances we may be able to deploy. The two primary latency costs are transport time (the cost of moving data between servers, CPU sockets and cores) and processing time. As all actions must finally be represented in a single state output, all action processing paths through the system must inevitably meet. Thus the interaction latency of the system is dependant on the aggregation path with maximum latency. Although parallelism reduces the processing latency along the processing path, it also increases the transport latency for aggregation by requiring increased movement of data between (and within) components. As opportunities for data reduction are reduced (less data to reduce on for each stage as the data has been partitioned), further layers of aggregation processing need to be added to finally bring all data together, increasing

\(^3\)The number of candidate values in the top \( N \) list is determined by the application implementation.
the amount of data shuffling. Modern processor architecture is memory bound (hundreds of processor cycles to access main memory) [22], [23], with the cost of network transmission orders of magnitude again. Therefore by minimising the amount of data movement through the system we can decrease the interaction latency, especially as the number of concurrent participants increases (the effectiveness of data reduction increases proportionally).

A. Properties

A primary driving force behind Concentus is maximising processing efficiency. To this end, we have applied the following properties to components in the system:

Spatial Locality: by keeping data as close as possible to processor cores, we maximise the effectiveness of the memory sub-system. The disparity between main memory access times and processor speed has resulted in a small amount of high speed memory being placed directly on the CPU die (CPU cache) [22]. Rather than reading data directly from main memory, all data accesses pass through this cache with cache misses resulting in a call out to main memory to populate the cache entry. Data is loaded into this cache from main memory in cache lines rather than as individual bytes (the cache line size on current processors is 64 bytes [22]). Thus when loading data from one memory address, we also get data from adjacent addresses (within the cache line) at no extra cost. Additionally, accessing data with a predictable stride pattern (i.e. moving through memory sequentially with the same address jump) allows prefetching optimisations to bring data into the cache ahead of time [24].

We apply this theme throughout the system by utilising data structures that are populated and consumed sequentially. In particular, we use preallocated circular buffers for all event queues in the system, and circular sliding windows for reliable messaging components. All data that is needed for processing in the system is stored in-memory.

Single Writer Principle [23]: sharing data mutation responsibility between threads works against the optimisations of the memory sub-system by requiring that processor cores co-ordinate with one another over slower levels of cache (or the socket interconnect if the processor cores are on different sockets), stalling the processors [24]. To maximise performance, we have divided the processing pipeline into stages. Each stage is executed by a single thread that has exclusive access to the data required for that stage. For communicating between the stages, we have used the Disruptor inter-thread messaging library [4] which itself has been designed around the single writer principle.

Preallocation: finally, to complement the spatial locality property and ensure that the garbage collector does not become a performance bottleneck, we preallocate memory for the eventing infrastructure (including wrapper objects for marshalling and unmarshalling data) in each component. Circular buffers are used prolifically throughout the infrastructure, with each buffer containing preallocated memory (fixed size from configuration) for incoming and outgoing data. By allocating the entire circular buffer upfront, we can obtain spatial locality properties for entries in the buffer (which will be read sequentially) by ensuring they are allocated together.

B. Topology

Concentus is implemented primarily in Java (with Scala used for one variant of the aggregation engine). We have used an event-based approach for the implementation, with events sent and received asynchronously.

The framework is designed to be distributed on a cluster, and is decomposed into the following node types (to avoid ambiguity, we refer to a node as an individual instance of an application component running on a server: the mapping between nodes and servers is generally one-to-one):

Client Handler: client handlers are responsible for handling client input events; and sending update events for shared state changes and effects. Each client is bound to a single client handler, with the handler storing a proxy for the client that is given a unique identifier within the system (we use a long (8 bytes) for the client ID, with the first 16-bits representing the client handler ID, and the remaining 48 bits the unique identifier within the handler). Each client is also allocated to an Action Processor node by the client handler, which handles the execution of actions against the shared state. Clients send input events every 100ms (configurable) to their client handler, with the client handler generating an update event in response containing the latest shared state update, and any relevant effects. Client input events are the entry point for actions, which are forwarded onto the action processor when received.

Action Processors: action processors are hosted by the aggregation implementation, and are responsible for transforming client actions into effects; storing the active effects for all clients that have been bound to it; and generating candidate values from the active actions when the aggregation engine requests that a tick be performed. Each action collector stores the timestamp for the next tick, and generates effects from incoming actions accordingly. As actions are processed, the created effects are sent back to the origin client handler in an ActionReceiptEvent. These effects are then forwarded onto clients as necessary. Each time a new tick is requested, the Action Processors provide an iterator for generating candidate values from the active effects (it is up to the aggregation engine as to how this iterator is used, though currently both implementations immediately populate a buffer which is then passed onto subsequent aggregation stages).

Aggregation Engine: the aggregation engine is the variable part of our framework. It is responsible for the timing of ticks, and the aggregation of candidate values into collective variables during each tick. It hosts both the Action Processor nodes (also determining the number of action processors deployed), and the shared state processor.

Shared State Processor: the shared state processor receives a Map object from the aggregation engine mapping collective
variable IDs to the associated collective variable states; and the
timestamp of the current tick. Each collective variable contains
the top N ranking of candidate values. On reception of this
map, the shared state processor generates a new state update
using the collective variables, the timestamp, and the previous
state. This update is then sent to all client handlers.

All nodes in the cluster are coordinated using Apache
ZooKeeper\(^7\) which we use for service advertisement and dis-
covery; and node configuration. A coordinator node (currently
a minimal implementation) sets up the cluster by starting up
nodes in the correct order and watching for dead nodes.

C. Processing Pipeline

Fig. 2 presents the pattern we have used for processing
components along the pipeline. We have applied the properties
outlined in the beginning of this section to the design of the
event flow. Before describing the event flow through the node,
we discuss the core components:

\[\text{ResizingBuffer}\]

Fig. 2. The general processing pipeline pattern used in Concentus.

\textbf{ResizingBuffer}: we have implemented a memory
buffer that is backed by a preallocated block of memory
of a fixed size (specified on the creation of the buffer).
The class exposes methods for directly reading and storing
primitives and strings to and from the backing memory, and
also provides methods for slicing the buffer (so that another
ResizingBuffer instance accesses into the backing array
at some offset). When data that is too large for the default
preallocated block is placed into the buffer, it temporarily
allocates a new memory block while retaining the preallocated
one. When the buffer is reset for reuse, it switches back to
the preallocated memory block, and frees the temporary one.
The rationale behind the buffer is the spatial locality and
preallocation properties. Multiple ResizingBuffers can
use memory that is spatially located by default, only switching
to alternative memory blocks when the preallocated memory
is too small (buffers should be sized to accommodate the
majority of messages). In the current implementation these
buffers are backed by standard Java byte arrays, though as
\text{ResizingBuffer} is an interface, this can be swapped
out with alternative approaches without changing event code
(e.g. allocating memory off heap).

\textbf{Disruptor}: as discussed in the \text{Properties}\subsection{Properties}
subection, the Disruptor is a lockless inter-thread messaging library. It con-
ists of a ring buffer filled with preallocated entries, and gating
logic that allows entries to be claimed and published in a
thread-safe and efficient way. We use the Disruptor as the
backbone data structure for the event processing pipeline, with
a single producer and consumer per Disruptor. We preallocate
the Disruptors with \text{ResizingBuffer} instances.

\textbf{Messaging}: for inter-node messaging, we use the 0MQ
\text{messaging library}\(^8\) which has been designed for low latency,
high throughput messaging (one of its core applications is
stock exchange messaging). The core abstraction in 0MQ is
the 0MQ Socket, which has a similar interface to Berkeley
sockets: raw bytes are sent and received with marshalling
(and unmarshalling) left to the application. Its advantages
include strong message delivery semantics: messages are either
completely delivered, or no bytes are received; performance
(bytes are only sent when the receiver is ready to buffer them);
and optimised messaging patterns.

\textbf{BufferBackedObject}: we use a custom
binary format for serialisation of events that allows
\text{BufferBackedObjects}, event reading and writing
objects, to directly index into the memory backing a
\text{ResizingBuffer}. Event processing code creates instances
of \text{BufferBackedObjects} for each event type they
process (e.g. \text{ClientInputEvent}) on instantiation; when
reading an incoming event they simply set the backing buffer
on the \text{BufferBackedObject} and read the fields. Likewise
when sending an event, they first claim a \text{ResizingBuffer}
entry from the send queue, set the backing buffer on the
\text{BufferBackedObject} of the required type, and write the
fields directly. This targets both the preallocation property
(not creating new objects so as to avoid copy costs), and the
spatial locality property (read data directly from the backing
buffer which has spatial locality with event before and after).

D. State Aggregation

We have developed two approaches for the aggregation
engine: one built on the Spark Streaming platform (a platform
for general purpose stream processing) \([10]\); and another that
performs all aggregation on a single processor.

\textbf{Spark Streaming Implementation}: the Spark Streaming
platform (which we will henceforth simply refer to as Spark)
has been designed for low latency stream processing \([10]\). It
is similar to other stream processing engines \([12]\) in that it
divides a computation over a cluster, processing unbounded
streams of data and emitting a stream of results. We are
interested in the applicability of general purpose stream com-
puting engines to online collective applications. Spark seemed

\(^7\)http://zookeeper.apache.org

\(^8\)We use version 3.2.4 of libzmq and version 2.2.2 of jzmq, the Java binding
for 0MQ. Both are available from http://zeromq.org/.
a particularly good fit as it processes streams in time-based batches (discretised streams), rather than as a record-at-a-time computation, fitting the tick approach of the application model.

To perform a computation on a running Spark cluster, users must supply a Driver application that defines a computation pipeline (using Spark’s API) consisting of transform operations (e.g. map, reduceByKey); and output operations (e.g. collect, foreach). Spark divides the computation into stages, which it distributes across the cluster. As the computation runs, Spark shuffles blocks of data between processing operations, finally bringing the output data stream back to the Driver application. The pipeline must start with one or more InputDStream endpoints that serve as the access point for elements into the computation; and end with output operations on the final stream(s) (e.g. writing the output elements to the network, printing the results to the console). To allow Spark to consume CandidateValues, we developed a custom InputDStream implementation called CandidateValueDStream that hosts an instance of an ActionProcessor. Instead of using our own tick timer, we intercept calls from Spark’s timer (when it attempts to generate the next batch for the stream), and forward these onto the ActionProcessor instances, which then return a batch of CandidateValues (using the time value from the Spark call). Spark allows multiple input streams to be created, which we leverage to allow multiple ActionProcessor instances to be deployed.

**Single Batch Sort Implementation:** though distributed general stream processing engines like Spark provide parallelism for stream computation tasks, they must also shuffle data between servers to achieve this parallelism. For example, to perform the top N ranking of CandidateValues, all matching values for some variable ID must eventually be collected together from all ActionProcessors in the system. As an alternative to the distributed approach, we have developed an implementation that simply sorts the entire batch of CandidateValues in order of the CollectiveVariable ID, data, and then score (using the default TimSort algorithm in Java 7); and iterate over the list aggregating as elements are encountered to create the final CollectiveVariable set. This approach only allows one ActionProcessor to be deployed in the system, but avoids moving data over the network (data instead is moved between threads using the Disruptor).

**VI. EVALUATION**

To demonstrate the viability of the framework for supporting online collective applications, we have evaluated its performance using the two aggregation engines. The intention of the experiment is to measure the performance of the framework in isolation, without consideration of additional performance aspects such as Internet latency. We deployed the framework onto the Amazon EC2 cloud computing platform along with an evaluation platform we have developed alongside Concentus. Load is generated by Test Worker nodes that simulate the required number of clients using agents provided by each experimental condition. For the evaluation, we measure the key metric of interaction latency: the latency from performing an action (on a client) to having that action perceptible in the shared state (i.e. the aggregated product reflects the effects of the action); against concurrent client count. Using the two conceptual applications discussed in the motivation section, we developed three application conditions that stressed the system in different ways. They are as follows:

**CrowdAloud!(Text):** this condition has one CollectiveVariable that collects the top 25 phrases submitted for the variable. Each phrase is freeform text, up to 140 unicode-16 characters in length. For two phrases to be identified as the same, they must match text bytes exactly: the grouping key is equal only on an exact match. Client agents submit phrases once a second (1Hz) with an almost uniform spread for phrase length[9]. This condition involves a substantial amount of data movement through the system (Actions, Effects, and CandidateValues store the full phrase data), though at a slower rate than other conditions. Effects have a duration of up to 5 seconds, with the score of each generated CandidateValue reflecting the duration the effect has been active (through a similar function as presented in the motivation).

**CrowdAloud!(Symbol):** this condition is identical to CrowdAloud(Text) with the exception that the phrase can be one of 32 symbols (e.g. emoticons), represented by a single integer (4 bytes).

**Collective Pong** this condition still uses one CollectiveVariable, but applies a sum aggregation on CandidateValues. The paddle bar is divided into 256 chunks, with each chunk representing the count of overlapping paddles at the associated paddle bar position. Individual paddles span a length of 8 chunks. Client agents submit actions containing the start position of their paddle once every 100ms, which are transformed into Effects storing this position as an integer. On the generation of CandidateValues, each individual paddle position is expanded out to a partial paddle bar: an integer array of length 256 with the chunks representing the individual paddle given a count of 1 and the remaining set to 0. All CandidateValues share the same group key, and so all end up reducing with the sum aggregation (by summing up the partial chunk counts) to generate a final paddle bar. This condition involves more data movement (increased action throughput to 10Hz), though the initial Action data size is small (one integer). However, CandidateValues are relatively large in size (256 x 4 bytes), and so involve the aggregation engine handling a large amount of data.

We deployed the framework using cc2.8xlarge cluster compute instances each with 2 Intel Xeon E5-2670 processors (with each processor having 8 physical cores, and 16 virtual cores with Hyper-Threading) and 60.5GB of RAM. We chose

9The phrase length is determined by a combination of the client ID and the current action sequence modulo 140.
10An individual paddle starting at chunk 3 will increment the count of chunks 3, 4, 5, 6, 7, 8, 9, 10 by 1.
Fig. 3. Mean latencies and jitter for processed actions for the two application variants (with CrowdAloud! having two modes) as the number of concurrent clients is increased. The tick duration is in the brackets beside each aggregation condition. With the exception of the Collective Pong condition, the percentage of late actions (actions that were lost or timed-out) was less than 3% (most less than 1%). We include the late action percentages for Collective Pong here.
to use cluster instances over standard instances to get access to the 10Gbps full bisection ethernet provided to these instances. Each node is deployed on a single server, and we limit each Java Virtual Machine instance (for hosting framework nodes) to 4GB of RAM. For all experimental conditions we use 4 Test Worker nodes and 4 Client Handler nodes. We test each application condition against the following three aggregation engine conditions (giving 9 experimental conditions):

**Single Batch Sort:** this implementation uses one additional server that hosts a single Action Processor node and a Shared State Processor node.

**Spark Multi Server:** spark can be deployed across multiple machines by deploying Spark Worker instances and a Spark Master instance across the cluster. For this condition we deploy 4 Spark Worker instances and 1 Spark Master instance on their own servers. We use a separate server for the Driver application which also hosts the Shared State Processor. We configured Spark (using the driver application) to deploy 4 Action Processor nodes onto each of its workers.

**Spark Single Server:** to measure whether moving data over the network adversely affected the latency, we also deployed a single Spark Worker instance, a Spark Master instance, and the application Driver onto one server. In this set up, only one Action Processor instance is hosted (like the Single Batch Sort condition).

Each test run simulated a fixed client count for 1 minute, after allowing for a 30 second start up period. Between test runs all framework components (with the exception of the test coordinator and a supervisor program that runs on each server) were restarted on a fresh JVM to ensure any JVM optimisations were not carried over between tests. Metrics were collected on the Test Worker nodes in 1 second windows, and forwarded onto the Test Coordinator for storage in a SQLite3 database.

### A. Results

Fig. 3 presents the mean latency and jitter (standard deviation of measured latencies) client agents recorded for each experimental condition. All latency statistics have a count of at least 500,000 values (significantly more for the higher client counts). We additionally collected metrics on the action throughput generated by the Test Workers, with all conditions reporting expected figures related to the client agents (i.e. the client count for the CrowdAloud! conditions, 10 times the client count for the Collective Pong condition). In addition to the latency, we also captured the number of actions that were lost or timed-out. The only experimental condition where this loss was significant was the Collective Pong condition, with all other conditions having late action percentages (number of actions lost over the total number) less than 3%. The lowest tick rate we were able to support for the Spark approach was 500ms, while the Single Batch Sort approach was able to support a tick rate of 100ms.

For the crowd aloud conditions (fig. 3a-3d) the Single Batch Sort (100ms) at 35,000 and Spark Multi Server (500ms) at 45,000 on the CrowdAloud! (Text) condition; and for Spark Single Server (500ms) at 30,000 and Spark Multi Server (500ms) at 35,000 on the Collective Pong condition. Rather than a single large outlier in the results, observing the latency over the collection window (we collect the latencies in 1 second buckets) indicates that the performance for these outlier test runs seems consistently lower than the client counts around them. Additionally the outliers have been observed for different client counts in other test runs, suggesting the issue might be related to the deployment environment. We wish to explore this in more depth in future experiments.

### VII. Discussion

On the whole, the results show that the framework is capable of supporting a large number of concurrent clients in relatively low latency interaction. This provides evidence for the viability of online collective applications from an architectural standpoint. The core framework components such as the Client Handler layer are able to keep up with the incoming action rate, and broadcast updates to all clients with low latency.

The two aggregation approaches provide evidence that keeping data closer to processing elements will generally offer better performance. The Single Batch Sort approach performed better than Spark when the amount of data was small despite only offering a single Action Processor; performing better in the CrowdAloud! variants, and up to 20,000 concurrent clients in the Collective Pong condition. One reason for this is to do with the reliability features Spark provides: Spark replicates data between its workers which can add significant overhead. For online collective applications, this reliability is not necessary as the data is ephemeral (the final state might be of interest, however, though this can be made reliable in other ways). The advantage of Spark over the Single Batch Sort approach is the ability to deploy more Action Processors: this advantage become evident in the Collective Pong condition when the client count grew larger than 20,000 (~20KB of data to be aggregated); the Spark Multi Server condition was able to maintain a low (less than 1.2%) action loss rate, while the other deployments (including the Spark Single Server) quickly
rose. What seems likely here, given that both the Spark and Single Batch Sort conditions had the same issue, is that the performance of the Action Processor becomes a bottleneck when the size of the data gets too large. Further work is needed here to ascertain the reasons for this limit, and whether it can be extended.

In terms of the range of applications that can be supported, the current experiment indicates that interaction is possible if latencies around 500ms (excluding Internet latency) are tolerable. For smaller latencies, reducing the data size that needs to be aggregated can help, as can reducing the amount of data that needs to travel between processors (or over the network). In future we would like to explore a larger cluster deployment to measure the upper limits of the framework, as well as explore additional stream processing engines such as Storm [12].

VIII. CONCLUSION

We have presented an application model for engaging the collective experience in an online environment. The collective experience, the experience of engaging with others who share an identity in a collective context; has been targeted by the model by necessitating that participants perform collective action together to have any agency in the application. To support applications derived from the model, we have developed Concentus: an exploratory framework for developing system architectures and applications that involve large numbers of people interacting simultaneously on a shared state. Concentus allows different implementations of aggregation engine to be evaluated in-situ with other components for handling client connections, and broadcasting updates. We have evaluated the performance of the framework using two aggregation approaches: one that uses a general purpose distributed stream computing engine (Spark Streaming), and another that instead works towards the principle of spatial locality (Single Batch Sort), by performing all aggregation on a single thread. The results show that both approaches are capable of supporting a crowd of 50,000 people in interaction with a latency of 500-1000ms. The Single Batch Sort approach generally performs better than the Spark Streaming one until the amount of data to be aggregated became too large (~20KB in the current experiment) at which point it appears the Action Processor component, responsible for generating the data to be aggregated on every tick, became a bottleneck. The distributed nature of Spark meant that it could deploy more Action Processors in parallel, and so keep up with the increase in data. The results provide evidence for the viability of online collective applications from an architectural perspective.

ACKNOWLEDGMENT

This work has been partially funded by Microsoft Research Asia, through the project “Crowd and the Cloud: Event Central Social Networking with Mobile Devices”.

REFERENCES