Computing and Informatics, Vol. 35, 2016, 852–869

EVOLUTION OF REACTIVE STREAMS API FOR CONTEXT-AWARE MOBILE APPLICATIONS

Przemysław Dadel, Krzysztof Zieliński

Department of Computer Science
AGH University of Science and Technology
Kraków, Poland
e-mail: {pdadel, kz}@agh.edu.pl

Abstract. This article describes the role of reactive streams concept as a core communication pattern in mobile-first applications and proposes directions for evolution of its classic API to better match mobile application requirements. By designing the selected examples of mobile applications, the authors evaluate the Reactive Streams API which is an increasingly accepted standard for asynchronous communication with back-pressure. This design is then assessed with regard to solution expressiveness and the ability to satisfy functional and non-functional requirements of the stated problems. It is observed that the used API does not allow for a context propagation from a mobile subscriber to a publisher so that the streamed data can be well adjusted to the variable reception context of a given mobile device. To address this issues, a context-aware variant of this API is proposed and it is demonstrated and discussed by presenting an alternative application design.

Keywords: Mobility, reactive manifesto, voluntary computing, context-aware computing, distributed systems

Mathematics Subject Classification 2010: 68-M14

1 INTRODUCTION

This article aims to describe the role of reactive streams concept as a core communication pattern in mobile-first applications and proposes directions for evolution of its classic API to better match mobile application requirements.

Reactive systems strive to be **responsive** by providing responses in a timely manner and detecting problems quickly. They are aimed to be **resilient** – they stay responsive in case of a failure. They should also be **elastic** so they stay responsive under changing workload. Being **message-driven** is the last but certainly not the least principle that is described in this manifesto. Actually, asynchronous message passing is a cornerstone of reactive systems that makes achieving the aforementioned goals feasible by encouraging communication patterns that ensure loose coupling, isolation, location transparency and provide relevant means to delegate errors as messages.

Work on Reactive Manifesto [5] has shown that the problems it addresses in building distributed systems are very pervasive and Reactive Stream API [7] has been proposed as a way to standardise how reactive software components should communicate.

The wide-spread acceptance of the 'reactive' trend [2] and its successful implementations by leading software vendors bring more scientific attention to this approach to test its applicability and extensibility in more specialized domains of distributed software development. Due to the increasing presence of mobile devices and existing challenges in building software systems cooperating with them, we have found that mobile-based applications will be a highly rewarding field of research.

In our work the Reactive Stream API is evaluated by a design of selected classes of mobile-centric distributed applications. We analyse how application of this API allows to satisfy both functional and non-functional requirements, how complex applications are as far as the implementation point of view is concerned and how fit they are for running on the mobile device. We point out the strength of such a solution as well as its shortcomings. Next, we provide an alternative design with the state-of-the-art Context-aware Reactive Stream API that we propose and discuss how it helps to alleviate the reported problems.

This article is structured in the following way: the next section provides a brief introduction to the problem domain, in Section 3 we introduce the Reactive Streams API and model two classes of applications with it. Section 4 shows an alternative design for the same use cases with our modified API. Finally we present a related work in Section 5, summarize our research in Section 6 and present our plans for future development in Section 7.

2 PROBLEM DOMAIN

Mobile devices are rich resources of contemporary internet. When such devices are in questions, one generally means smart-phones, tablets, smart-watches and other portable, programmable appliances with human facing interface that can communicate wirelessly.

To properly build a distributed system in which mobile components play a significant role we need to take into account several factors. The first one would be

inherent, limited available resources, for example the battery power and the variable network connectivity. Heterogeneity is another one. In our work we focused on the Android ecosystem as an open, mature and most widespread platform in the mobile world. The applications we have developed in our study could work on over 2 000 different devices. Regardless of the software API differences of the Android Platform, those devices are equipped with a diverse sensor set and present a wide spectrum of raw computing power from very slow budget single-core devices to a multi-core top-level smart phones.

Mobility does not have to be perceived as a restriction, it can be also a feature enabler. Depending on sensors and available context data, devices can provide context dependent functionality (e.g. position, velocity, vicinity of other appliances) and this fact enables new classes of applications.

In our design experiment we chose two classes of mobile application that should benefit from the reactive approach and use of Reactive Streams API and whose behaviour is likely to be dependent on the properties that change due to the device mobility.

The first class consists of mobile applications that are end consumers of a data stream: e.g. traffic information, video, news feed, monitoring data and users (or business model behind the application) expects this stream to be up-to-date and adjusted to the current context of a user or device, e.g. location, currently played song or vicinity of other users.

The second class consists of a more peculiar group of applications where operations executed by a mobile device contribute to a more complex, externally orchestrated process and the device owner is not necessarily an active user of applications. An example would be collaborative types of applications for data collecting (e.g. traffic density), distributed offloading of operations in *ad-hoc* networks or a class of voluntary computing applications on mobile devices. In such cases mobile application requests a work stream that is executed on a device.

In our use cases we focus solely on the design of the mobile side of the software system.

3 REACTIVE STREAM API

The Reactive Streams API [7] is a rapidly being accepted programming interface with a couple of implementations in popular Java-running libraries e.g. Spring Reactor or Akka Streams. It allows software designers and developers to express and program communication between distributed participants that would enable 'reactive' characteristics of the application. Fundamentally it is a publisher-subscriber based communication with back-pressure. While API does not enforce such implementation, it allows to express completely asynchronous interaction between the participants so, in no case, one communicating side waits for another.

This fact is especially important when one builds a system which involves mobile devices. The reception and processing of a message sent to a mobile component may

be delayed due to network connectivity and availability changes, battery depletion and performance changes caused by e.g. OS level scheduling that aims at providing maximum performance to the front user actions and preserving energy while a given user does not actively use a particular device.

A standard interaction between participants (Figure 1) using this API (Listing 1) can be described in the following steps:

1. using a *Publisher* object reference a *Subscriber* subscribes and receives subscription confirmation through a callback method *onSubscribe*,
2. a subscriber receives a *Subscription* object through which it can *request* for next n-elements to process, data is provided through a callback method *onNext*,
3. communication ends either by explicit Subscription cancellation or data stream completion or error: *onComplete*, *onError*.

```
trait Publisher[Signal] {

  def subscribe(subscriber: Subscriber[Signal])
}

trait Subscriber[Signal] {

  def onSubscribe(subscription: Subscription[Signal])
  def onComplete()
  def onNext(element: Signal)

  def onError(err: Throwable)
}

trait Subscription[T] {

  def cancel()
  def request(n: Long)
}
```

Listing 1. Reactive Stream API interfaces in Scala

3.1 Class A – Reactive Evacuation Application

To represent this class we chose an application for which the main functional requirement is to guide a user through a walking route with a variable traffic congestion and traversal plan. One can think of an evacuation route e.g. out of his office, event venue or a sightseeing plan in a crowded museum. While a need to continuously evaluate a route on a mobile phone's screen would be a severe usability limitation, an increase in popularity of wearable appliances e.g. smart-watches or smart-glasses, makes such information much more accessible to the user. To the contrary of a simple navigation application, this variable traffic and plan case puts more emphasis on the relevance of a current user/device context.

In the designed application high-level logic would look the following way:

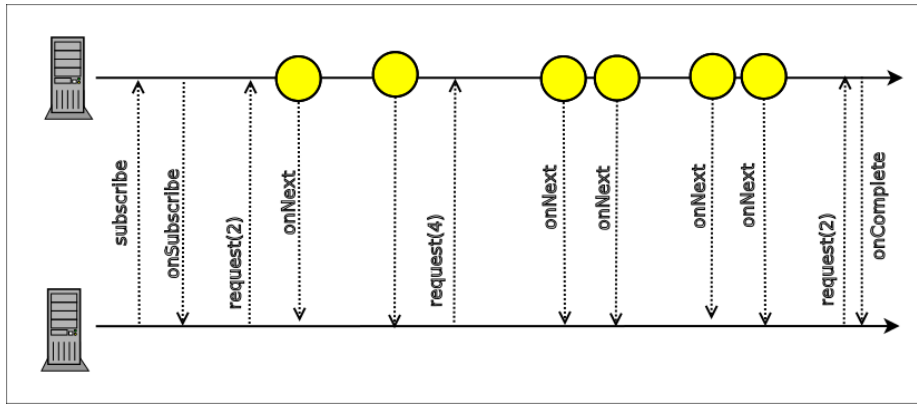

Figure 1. Publisher-subscriber interaction representing situation of context expiry or reconnection

- a user or an alarm signal controls when to start the emergency data delivery,
- the mobile application subscribes to an evacuation service (publisher) for the next positions giving its start location,
- the navigation service (reactive stream publisher) provides a subscriber with a stream of next-move commands,
- the mobile application controls the rate of those commands using the *request* method on subscription.

The designed application will track the user movement through its location service and adjust the rate of back-pressure request calls to the speed of location changes. The application is then not flooded with data as it requests them when needed.

The current API, however, does not allow the mobile application to report back on its location and as result next-move commands with time may lose synchronization from real users movements. One could argue that a given device knows its location and how much it deviates from the guided route but there may be a good reason for choosing a different path e.g. a blocked passage or other unexpected obstacles. We could alleviate this by re-establishing a subscription after a time-out (function of user's velocity) or when the user gets further and further from the planned route. However it makes the implementation complicated, forces application to be an active communicating side (in contrast to only reacting to server initiated on *Message* calls) and only temporarily mitigates a stale data problem.

There are other valid scenarios hard to achieve with current API. Firstly, mobile devices are equipped with various sensors and an example of adding temperature measurement to location information would allow navigation service to better adjust the escape route e.g. in the case of a fire. Secondly, in the case of evacuation, more users are likely to take part in it and live 'feedback' on individual users relocations

and sensor data would allow a navigation service to better adjust an evacuation plan for each user to, for example, minimize bottlenecks or detect blocked paths faster.

Outstanding problems:

- adjusting the rate of incoming messages to the users current context (e.g. the hotter the faster route is updated),
- delivery of information relevant from the perspective of the current user context,
- ability for a producer to adjust streamed data to the current context of multiple subscribers,
- complex administration required to retain relevant subscription,
- back-pressure only on volume but not on the stream parameters.

We will present how these limitations could be avoided with the proposed Context-aware Reactive Stream API.

3.2 Class B – Mobile Voluntary Computing Application

This application aims to use the power of a mobile device while it is not actively being used. BOINC[3] is an example of the already existing projects that have a mobile client[6] that executes computational tasks on a mobile device when it is on Wi-Fi and is connected to a power supply.

With this use case we will focus on how to provide a stream of work to the device that matches best its capabilities.

In the designed application high-level logic would look in the following way:

- a user installs a mobile application that can execute computation tasks as background process,
- the application subscribes to the publisher for a stream of tasks,
- the application describes the profile of the device (e.g. computation power) during subscription,
- the computation orchestrator (stream publisher) provides a subscriber with a stream of tasks,
- the application requests n-next tasks if it can process more of them (e.g. is on Wi-Fi and is powered on),
- the application controls a task rate using request(n) method on subscription.

The Reactive Streams API's back-pressure mechanism plays a vital role in this type of application: it protects a mobile computation unit not only from being flooded with data but also lets it to be responsive as it will only process tasks when it is able to do so. This also protects the users' Service Level Agreement (SLA) under which they agree to contribute their device to the programme.

In this case we also notice a stale data problem but it is not so crucial as in the first example. If we provide tasks with enough fine grained granularity, the impact

on the device should be minimized if, at reception, it turns out that e.g. the device is running back on battery.

However, there are other, more relevant problems that are not so easily handled with the current API. The first is a situation when a change of device properties would not disqualify the device from contribution but would rather require different kind of tasks. A device low on battery, with reduced operational memory or with worse network connectivity would still be able to responsively process the task if these tasks would be suited for device condition e.g. split. Another example would be tasks that should be only executed when a device is connected to a particular network location to benefit from data locality. It is somewhat similar to the stale view problem and it also could be alleviated with a re-subscription mechanism but a burden of state tracking and subscription management remains on a mobile device.

The second problem is the publisher's perception of a device state in long running processes and handling the device reconnection. On a regular basis we would expect that computational tasks are delivered to the device soon after a *request(n)* is invoked, but it may happen that the orchestrator has currently no tasks that match device capabilities. Those tasks will be delivered to the device as soon as they show up and a mobile application has previously expressed the desire to consume more data. However at that point the *request(n)* may be long based on an outdated state of device. The producer has no information on how long it can rely on the subscriber's *request(n)* call and has no way to express that the subscriber's request has expired.

Additionally, a subscribed device may simply get deactivated or go offline so the producer needs a way to 'wake up' a subscriber so that it can repeat its request. Subscribers could try to actively track its relevant context and re-establish subscription but a leaner approach would be to let the device be as passive as possible and orchestrator can take in turn responsibility of ordering re-request(n) or re-subscription from the device.

Outstanding problems:

- delivery of content matching current mobile device capabilities,
- long running subscriptions and maintaining an up-to-date perception of device properties,
- refreshing subscribers after periods of silence.

4 CONTEXT-AWARE REACTIVE STREAM API

In our design experiment we have found that Reactive Stream API is a great tool to build mobile clients that require for content to be pushed like a stream to the device. The ability to back-pressure plays an important role to ensure that a device receives only the load it can handle and when that data is needed.

We observed that Reactive Stream API addresses the problem of adapting quantitative parameters of the data stream. It is useful for systems with fixed characteristics (e.g. computing power, broadband network, constant power supply) to be

elastic under a changing load and with back-pressure they process only the amount of data they claim they can.

However, characteristics of the mobile running consumers change over time. The **context** under which mobile subscribers process data is variable. We find that a solution to the design problems pointed in the previous sections would be that the mobile subscribers could back-pressure on qualitative parameters of the data stream under the same subscription.

To address that need, in this article an extended **Context-aware Reactive Stream API** is presented. One can enumerate the following key differences:

- communicating parts are parametrized not only with the type of streamed data but also the context information that may influence the stream,
- while requesting next elements to process, the current device context is passed to a publisher,
- the context has its validity period and the publisher notifies the subscriber with *onContextExpired* method when the context expires.

Presence of this *onContextExpired* extensions is a consequence of three factors:

- context is valid for a period of time,
- mobile clients are supposed to be passive (they do not spontaneously request demand),
- mobile clients are expected to get out of signal, lose connectivity and they may not be aware of being out producer's reach – we want a subscription to hold even in such cases. This contributes much to resilience of this communication strategy.

While we could not provide binary compatibilities between APIs we can notice that existing Reactive Streams API can be perceived as a specialization of our proposal if we introduce an *Empty* context that never expires and leave onContextExpired implementation empty.

```
trait Publisher[Signal, Context] {

  def subscribe(subscriber: Subscriber[Signal, Context])
}

trait Subscriber[Signal, ContextType] {

  def onSubscribe(subscription: Subscription[Context])
  def onComplete()
  def onNext(element: Signal)
  def onContextExpired(ctx: Context)

  def onError(err: Throwable)
}

trait Subscription[Context] {

  def cancel()
  def request(n: Long, ctx: Context)
```

```
}

trait Context {
  val expireAt: Int
}
```

Listing 2. Context-aware Reactive Stream API interfaces in Scala

Examples of context that can affect subscriber capabilities might be among others:

- the position of the device – if the device position makes the device ineligible for content, stream could be closed, but we may find applications with a requirement to provide e.g. more customized/enriched context when the subscriber is in the vicinity of a certain location,
- a change of the network speed by impact quality of the transmitted data e.g. stream of video, images, synchronization depth,
- a device on battery – move intensive tasks to execute,
- higher blood pressure – adjust stream of music to calm down the user.

The presented API allows developers to express communication between distributed participants and update producer on the context of data reception resulting in context back-pressure. Contrary to the situation when a mobile application itself manages validity of the subscription, with our approach a given mobile device can stay very passive and only takes action on demand and the demand is mostly controlled by the subscriber itself. The only exception are *onContextExpired* calls that are publisher initiated (but context time-out is still subscriber defined).

The streamed data may not be dependent only on this current device context, in a collaborative kind of mobile application data may change depending on the context of other devices. When context information pushed up the data stream, the publisher is allowed to make use of this data.

4.1 Class A – Context-Aware Reactive Evacuation Application

In this section, the design and code samples of a context-aware version of the previously proposed solution will be presented for the updated API. For the simplicity of code examples we use the location context only.

In the designed application high-level logic would look the following way:

- a user or alarm signal controls when to start the emergency data delivery,
- the mobile application subscribes an escape service (producer) for the next positions giving its current location,
- the mobile application requests next n move-commands using *request* method on subscription and provides current location with each request.
- the navigation service (reactive stream publisher) provides a subscriber with a stream of next-move commands adjusted to current reported context and reported locations of other users in the area.

The presented data flow and code sample indicate that adhering to this API can result in a simple, expressive subscriber implementation. We achieve a goal of a clean design that allows the application to be responsive both from the communication and the user reception perspective.

```
class EvacuationApp extends Subscriber[Move, Location] {

  var subscription: Subscription[Location] = _

  def start() {
    NavigationPublisher.subscribe(this)
  }

  def currentContext(): Context[Location] = {
    Context(expireAt = fromNow(60 sec), state = Location(fromGPS()))
  }

  def onSubscribe(sub: Subscription[Location]) {
    subscription = sub
    // request next position
    subscription.request(1, currentContext())
  }

  def onComplete() {
    println("You are safe now.")
  }

  def onNext(move: Move) {
    println("Go to: " + move)
    // two more steps
    subscription.request(2, currentContext())
  }

  override def onContextExpire() {
    subscription.request(1, currentContext())
  }

}
```

Listing 3. Context-aware Mobile Evacuation Application

4.2 Class B – Context-Aware Mobile Voluntary Computing Application

By redesigning a mobile voluntary computing application attention is paid to a different aspect of the proposed API. While the context back-pressure is highly relevant is this case, the importance of context validity management and maintaining a stream in volatile connectivity environments is even more visible in this class of applications.

In the designed application high-level logic would look the following way:

- a user installs a mobile application that can execute computation tasks as background action,
- the application subscribes to the producer for a stream of tasks,
- the application describes a profile of the device (e.g. computation power) during subscription,

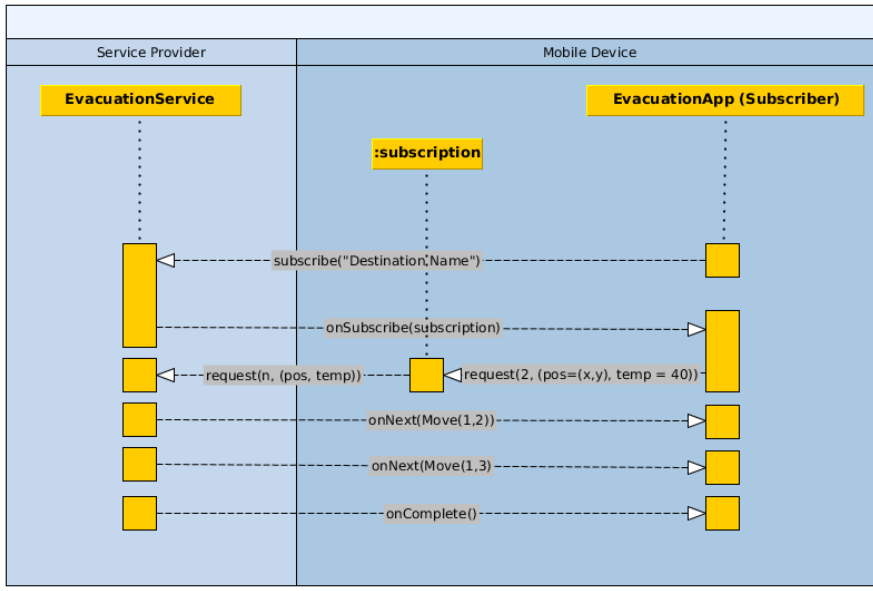

Figure 2. Sequence diagram for evacuation service

- the application requests n-next tasks if it can process more of them (e.g. is on Wi-Fi and is powered on) and provides its current context (e.g. free memory, connectivity type),
- the computation orchestrator – stream publisher – provides a subscriber with a stream of tasks adjusted to its profile and the reported context,
- in case of absence of matching tasks, the orchestrator will track validity of the current context and report that it has expired to get fresh application demand,
- *onContextExpired* is also sent after a period when the application's demand is not reported, this allows an application to re-establish its demand in the case when the device was restarted or temporarily lost connectivity.

Figure 3 represents a publisher-subscriber interaction when context expires or reconnection is needed and code sample (Listing 4) demonstrates what the implementation would look like.

```
class VoluntaryComputingApp extends Subscriber[Move, ProcessingCtx] {

  val executor = new TaskExecutor()

  def start() {
    TaskProducer.subscribe(this, initialContext())
  }

  def onSubsribe(s: Subscription) {
```

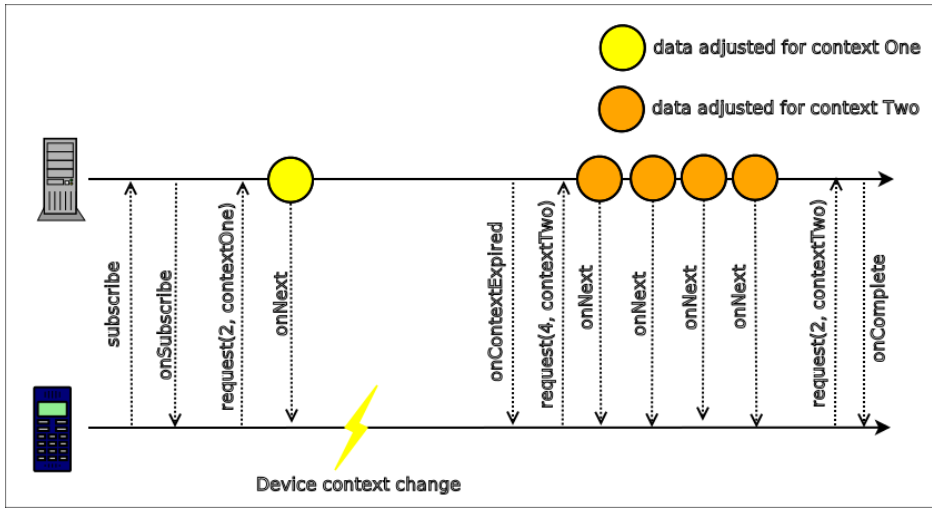

Figure 3. Publisher-subscriber interaction representing situation when context expires

```
    this.s = s;
    s.request(nextTasks(), currentContext(timeToLive));
  }

  def onNext(nextTask: Task) {
    executor.process(nextTask);
    s.request(nextTasks(), currentContext(timeToLive));
  }

  def onContextExpired() {
    s.request(nextTasks(), currentContext(timeToLive);
  }

  private def nextTasks(): Int {
    if (wifi = true && power = true) return 1 else return 0;
  }

}
```

Listing 4. Context-aware voluntary computing worker

Both examples presented to contrast previous software design showed that the gap between a context-aware publisher and a context-aware consumer can be filled by a communication mechanism based on Context-aware Reactive Stream API (Figure 4).

4.3 Context-Aware Publisher Considerations

In this work, the authors focus on the use of the API from the client requirements and the implementation perspectives. By following a bottom-up, use-case driven design and starting with a client, the authors try to validate suitability of the Context-

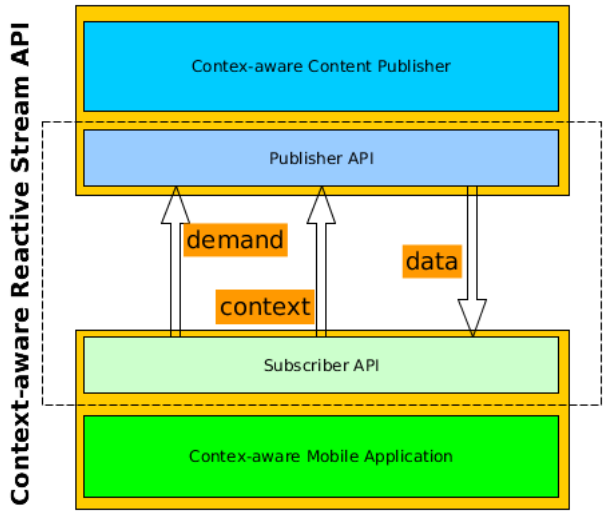

Figure 4. Context-awareness in a software system requires for the context to be propagated across communication API

aware Reactive Streams approach especially in the parts of the system where its owners have less control over – mobile applications. Usually, stream publisher will reside on an application back-end server which is easer to administer and tune in order to match the scalability requirements. Actually with the Reactive Streams API we are more advantaged with ability to predict and handle a server load:

- client request n-items that are delivered back asynchronously by server – in case of an increased load server can produce data slower or auto-scale,
- server "knows" when it can expect further demand – in response to onNext message,
- ability to track context and make a load predictions based on it adds yet another dimension to consider.

While Reactive Streams API itself does not define it, its implementations e.g. RxJava, Akka Stream allow functional transformations of a Publisher by applying, among others, *map*, *filter*, *concat* functions. The authors are aware that these operations may not be possible or may not have the same semantics with a less restrictive context-aware generalisation of this approach. However, the most important responsibility of the Reactive Streams API is governing an asynchronous directional channel with back pressure and this responsibility is satisfied.

The authors argue that the context propagation is vital on the mobile-server boundary. The client-side code can eventually narrow the context-aware stream to a simple data stream (Listing 5). As for the context propagation on a server-side,

in the current prototypes we assumed that the context is passed to the actual data source and we did not consider any transformations possible on it.

```
val ctxAwarePublisher: CtxAwarePublisher[Task] = ...
val regularPublisher = ctxAwarePublisher.withContextEnrichment( (
    ctxAwareSubscription) -> new Subcription {

  def request(n: Int) {
    ctxAwareSubscription.request(n, computeContext());
  }

})
```

Listing 5. Narrowing of a client's stream view

The code samples (Listings 6 and 7) demonstrate very high-level suggestions on how context-aware publisher could be implemented for both presented applications.

```
class EvacuationMovePublisher extends Publisher[Move] {

  // dynamically updates routes depending on position of all subscribers
  val source: MoveSource = ...

  private def request(n: Int, ctx: Context, subscriber: Subscriber) {
  source.updatePosition(subscriber, ctx.position)
  source.routeForSubcriber(subscriber, ctx)
        .take(n)
        .to(subscriber)
  }

}
```

Listing 6. Context-aware evacuation route publisher

```
class VoluntaryComputingTaskSource extends Publisher[Task] {

  val source: TaskSource = ...

  private def request(n: Int, ctx: Context, subscriber: Subscriber) {
  source.select( task -> matchContext(task, ctx)
        .take(n)
        .to(subscriber)
  }

}
```

Listing 7. Context-aware voluntary computing task publisher

For current considerations, details of how a stream publisher would need to handle and use the context information are not discussed thoroughly as this depends much on the problem being solved. One could argue that this information can increase back-end side complexity. It is likely to be true but context-awareness is not something optional – it is an enabler for certain classes of applications. It means it is an inherent complexity and the presented API aims to minimize accidental complexity at client side by providing a clean programming model. Nevertheless, the authors plan to develop this concept further and propose a set of patterns to help reduce this complexity at publisher's side.

5 RELATED WORK

The concepts of Reactive Programming, Reactive Systems and Reactive Systems are subject of an ongoing and growing interest of software development practitioners and researchers. The reactive programming walk-through [13] indicates that it has roots in functional programming. This style of programming has been demonstrated [12] in implementing user interfaces (UI) as it provides clean programming model for dealing with user triggered events. In the recent years Reactive Systems have been seen as a clean model of designing and building a scalable distributed system and the work on the commercial Typesafe Reactive Platform [14, 2] demonstrates power of reactive programming in enterprise class distributed applications.

The study on "Context-Aware Mobile and Wireless Networking" [9] summarizes taxonomy and current challenges in building context-aware software that is well integrated across networking and computing environments. It calls for "standardization initiatives dealing with scalability and interoperability issues in a future context ecosystem" and "greater contextualization of communication services supporting both context-triggered actions and context-dependent reactions". These problems are explicitly addressed in our work.

A number of projects have already aimed to solve the problems in communication, integration and management in mobile-based systems. Rx4DDS.NET [8] addresses challenges in data distribution by integrating data-centric publish/subscribe technologies, such as Object Management Group (OMG) Data Distribution Service that handles distribution of data to interested subscribers with The Reactive Extensions (Rx) library [1]. This reactive library is similar in many ways and conceptually compatible with the Reactive Streams API. Rx4DDS.NET allows subscribers to conveniently compose and transform streams (clean programming model) and the functional nature of composing operations enables scaling-up of an end-processing node. This solution is validated against the problem of processing streams of sensor data and emphasises the validity of reactive approach in Internet of Things (IoT) applications. Even though a somewhat reverse situation is described in our paper, as in our case mobile (IoT like) devices are subscribers, the reactive streams approach is highlighted as a suitable technique in building clean and maintainable software for Internet of Things (IoT).

Ambient Clouds [11] is a project for representing collections of remote objects in MANET networks that combines event-driven interaction, bases on publish/subscribe model with reactive programming constructs: e.g. mapping, grouping concatenations, filtering. The result collection is transiently updated in the case of changes in aggregated remote objects. While the nature of the used data structure is different: bounded collection vs. unbounded stream we could model similar constructs with our API if we subscribe to a stream of collection updates and notice that reactive operations on a stream are provided by most reactive stream libraries. Context back-pressure could be used here by reactive operators to control granularity of collection updates e.g. filter operation would push information up the stream so that filtered property changes are propagated faster.

As far as robustness and communication efficiency are concerned, the work [10] on Erlang-based sensor management systems has similar focus as the proposed API: provide fault tolerance and increase the relevance of the transferred data. Those aspects are emphasised with actor-based processing model as a mean to handle sensor's failures and to pre-process data by sensors locally. Our focus is put on a communication strategy that expresses a way to preserve battery and throughput (by minimizing pulling of data) and assumes communication faults in the system. We perceive that both solutions could be combined in a similar way as Akka Streams bridges classic Reactive Streams API with Akka – the actor model implementation for Java Virtual Machine.

Work on CEMCloud [4] provides an excellent argument for testing boundaries to which we can use mobile device as a worker by proving that mobile devices can be a source of a very energy efficient computing power. What we strive to provide is an equally efficient way of providing work to them.

6 CONCLUSIONS

Context-awareness is a crucial requirement that has to be addressed at the core of architectural patterns used to build mobile applications. The Reactive Streams API has a lot of advantages and enables reactive distributed communication, but we have discovered that it is not open enough to fully bring reactive principles to mobile clients. We have discussed examples when the user's current context is a key parameter that should be used to back-pressure not only on volume but also on quality and content selection of streamed data. We discovered that the stream publisher for a mobile device has to be a lot more agile and allow for qualitative modification of parameters of data streams depending on device **context**. e.g. lowering streamed video quality for battery worn-out device.

To address these issues we have analysed existing standard of the Reactive Streams API and used it as a basis to build a new Context-aware Reactive Stream API. It allows mobile clients to request data for specific context of data reception so that the publisher can adjust the streamed content accordingly. The specified context has its validity period, so the publisher can determine how long it can rely on this request and when it needs to ask the subscriber to refresh its demand.

Additionally, the presented API is much simpler to use from the perspective of the mobile application implementation as the only state it needs to preserve is an open subscription. A mobile application is off-loaded from complex subscription monitoring and context tracking so that most of the application structure can be devoted to the core domain implementation.

The cost of this approach is a more complex publisher implementation, but this complexity is not accidental. It comes from the nature of the used components, e.g. it is better to offload a mobile device from unnecessary responsibilities and it is also a property of the class of the problem solved, e.g. group navigation services or computation orchestration services are inherently stateful.

7 FUTURE WORK

In this paper we have discussed an API proposal that is backed up by several designed use cases. A prototype implementation has been provided so far for the second – computational – use case and the authors would like to experiment more with real life implementations. From a different perspective, we have optimized for the simplicity of the mobile subscriber and we would also like to investigate more and provide guidelines for effective implementation of context-aware publishers. We plan to close our research on reactive streams in a mobile world by investigating the role of mobile devices as data publishers as well as the mechanisms which would allow to subscribe to a population of mobile devices, for example to get a heat map or traffic density. We find these topics highly relevant due to the visible progress both in the adoption of mobile devices and their capabilities.

REFERENCES

[1] The Reactive Extensions (Rx). http://msdn.microsoft.com/en-us/data/gg577609.aspx.

[2] Typesafe Case Studies. http://www.typesafe.com/resources/case-studies-and-stories.

[3] Anderson, D. P.: BOINC: A System for Public-Resource Computing and Storage. Proceedings ot the Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10.

[4] Ba, H.—Heinzelman, W.—Janssen, C.-A.—Shi, J.: Mobile Computing – A Green Computing Resource. 2013 IEEE Wireless Communications and Networking Conference (WCNC), IEEE, 2013, pp. 4451–4456.

[5] Boner, J.—Farley, D.—Kuhn, R.—Thompson, M.: The Reactive Manifesto. http://www.reactivemanifesto.org/.

[6] BOINCs Homepage. BOINC on Android. http://boinc.berkeley.edu/.

[7] Pivotal, Red Hat, Twitter, Typesafe, Kaazing, Netflix. Reactive Stream API. http://www.reactive-streams.org/.

[8] Khare, S.—An, K.—Gokhale, A.—Tambe, S.—Meena, A.: Reactive Stream Processing for Data-Centric Publish/Subscribe. Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS '15), ACM, New York, NY, USA, 2015, pp. 234–245.

[9] Makris, P.—Skoutas, D. N.—Skianis, C.: A Survey on Context-Aware Mobile and Wireless Networking: On Networking and Computing Environments' Integration. Communications Surveys Tutorials, IEEE, Vol. 15, 2013, No. 1, pp. 362–386.

[10] Niec, M.—Pikula, P.—Mamla, A.—Turek, W.: Erlang-Based Sensor Network Management for Heterogeneous Devices. Computer Science, Vol. 13, 2012, No. 3, pp. 139–151.

[11] Pinte, K.—Lombide Carreton, A.—Gonzalez Boix, E.—De Meuter, W.: Ambient Clouds: Reactive Asynchronous Collections for Mobile Ad Hoc Network

Applications. In: Dowling, J., Taïani, F. (Eds.): Distributed Applications and Interoperable Systems. Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7891, 2013, pp. 85–98.

[12] Prokopec, A.—Haller, P.—Odersky, M.: Containers and Aggregates, Mutators and Isolates for Reactive Programming. Proceedings of the Fifth Annual Scala Workshop (SCALA '14), ACM, New York, NY, USA, 2014, pp. 51–61.

[13] Salvaneschi, G.—Margara, A.—Tamburrelli, G.: Reactive Programming: A Walkthrough. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), May 2015, Vol. 2, pp. 953–954.

[14] Typesafe. An Introduction to Reactive Streams, Akka Streams and Akka HTTP for Enterprise Architects – White Paper. Typesafe Whitepaper. `https://info.typesafe.com/COLL-20XX-Enterprise-Architect-Akka-Streaming-Guide_TY-RP.html`.

Przemysław Dadel is a computer science Ph.D. student at the AGH University of Science and Technology. He pursues his interest in distributed, cloud and mobile computing. He is an experienced software engineer with over 5 years of commercial experience in Java and related technologies.

Krzysztof Zieliński is Head of the Department of Computer Science at AGH University in Krakow. His research concentrates on networking, mobile and wireless systems, distributed computing, and service-oriented distributed systems engineering. He is the author of over 200 papers in these topic areas. He was the Project/Task Leader of numerous EU-funded projects, e.g. PRO-ACCESS, 6WINIT, Ambient Networks. He served as an expert for the Ministry of Science and Education. Currently, he is leading an SOA oriented research performed by the IT-SOA Consortium in Poland. His research interests evolve around adaptive SOA solution stack, services composition, service delivery platforms and methodology. He is an active member of IEEE, ACM and The Polish Academy of Sciences. He served as a program committee member, chairman and organizer of several international conferences including MobiSys, ICCS, ICWS, IEEE SCC and many others.