Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management

Alexander Artikis[1], Matthias Weidlich[2], Francois Schnitzler[3], Ioannis Boutsis[4], Thomas Liebig[5], Nico Piatkowski[5], Christian Bockermann[5], Katharina Morik[5], Vana Kalogeraki[4], Jakub Marecek[6], Avigdor Gal[3], Shie Mannor[3], Dermot Kinane[7] and Dimitrios Gunopulos[8]
[1]Institute of Informatics & Telecommunications, NCSR Demokritos, Athens, Greece,
[2]Imperial College London, United Kingdom, [3]Technion - Israel Institute of Technology, Haifa, Israel,
[4]Department Informatics, Athens University of Economics and Business, Greece,
[5]Technical University Dortmund, Germany, [6]IBM Research, Dublin, Ireland,
[7]Dublin City Council, Ireland,
[8]Department of Informatics and Telecommunications, University of Athens, Greece
a.artikis@iit.demokritos.gr,m.weidlich@imperial.ac.uk,francois@ee.technion.ac.il,
mpoutsis@aueb.gr, {thomas.liebig, nico.piatkowski}@tu-dortmund.de,
{christian.bockerman,katharina.morik}@cs.uni-dortmund.de, vana@aueb.gr,
jakub.marecek@ie.ibm.com, {avigal@ie, shie@ee}.technion.ac.il,
dermot.kinane@dublincity.ie, dg@di.uoa.gr

ABSTRACT

Urban traffic gathers increasing interest as cities become bigger, crowded and "smart". We present a system for heterogeneous stream processing and crowdsourcing supporting intelligent urban traffic management. Complex events related to traffic congestion (trends) are detected from heterogeneous sources involving fixed sensors mounted on intersections and mobile sensors mounted on public transport vehicles. To deal with data veracity, a crowdsourcing component handles and resolves sensor disagreement. Furthermore, to deal with data sparsity, a traffic modelling component offers information in areas with low sensor coverage. We demonstrate the system with a real-world use-case from Dublin city, Ireland.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Systems—*query processing, rule-based databases*

1. INTRODUCTION

The recent development of innovative technologies related to mobile computing combined with smart city infrastructures is generating massive, heterogeneous data and creating the opportunities for novel applications. In traffic monitoring, the data sources include traditional ones (sensors) as well as novel ones such as micro-blogging applications like Twitter; these provide a new stream of textual information that can be utilized to capture events, or allow citizens to constantly interact using mobile sensors.

Detecting complex events from heterogeneous data streams is a promising vehicle to support applications for monitoring, detection and online response [11, 20]. Consider e.g. an urban monitoring system that identifies traffic congestions (in-the-make) and (proactively) changes traffic light priorities and speed limits to reduce ripple effects. Such a system may use traffic flow and density information measured by fixed sensors mounted in selected intersections, together with reports from public transport vehicles (buses, trams, etc).

Our work is motivated by an existing traffic monitoring application in Dublin City, Ireland. We present the general framework of a system that has been designed in this context, and the challenges that come up from a real installation and application. The long term goal of the related INSIGHT project[1] is to enable traffic managers to detect with a high degree of certainty unusual events throughout the network.

We report on the design of a monitoring system that takes input from a set of traffic sensors, both static (intersection located, traffic flow and density monitoring sensors) and mobile (GPS equipped public transportation buses). We explore the advantages of having such an infrastructure available and address its limitations.

Some of the main challenges when dealing with large traffic monitoring data streams are that of veracity and sparsity. Data arriving from multiple heterogeneous sources, may be of poor quality and in general requires pre-processing and cleaning when used for analytics and query answering. In particular, sensor networks introduce uncertainty into the system due to reasons that range from inaccurate measurements through network local failures to unexpected interference of mediators. While the first two reasons are well recorded in the literature, the latter is a new phenomenon that stems from the distribution of sensor sources. Sensor data may go through multiple mediators en route to our systems. Such

[1]www.insight-ict.eu/

EDBT-2014 Athens, Greece

mediators apply filtering and aggregation mechanisms, most of which are unknown to the system that receives the data. Hence, the uncertainty that is inherent to sensor data is multiplied by the factor of unknown aggregation and filtering treatments. In addition, data present a sparsity problem, since the traffic in several locations in the city is either never monitored due to lack of sensors, or infrequently monitored (e.g. when a bus passes by).

In [3], we outlined the principle of using variety of input data to effectively handle veracity. Streams from multiple sources were leveraged to generate common complex events. A complex event processing component matched these events against each other to identify mismatches that indicate uncertainty regarding the event sources. Temporal regions of uncertainty were identified from which point the system autonomously decided on how to manage this uncertainty.

In this paper we present a holistic view of traffic monitoring; we present approaches to address (i) the **veracity** of the data problem, (ii) the **variety** of the data problem, and (iii) the **sparsity** of the data problem. In addition, the streaming architecture we develop is scalable, and therefore capable of addressing the **volume** of the data problems that arise as the available data sources increase. We integrate the respective techniques in the context of a unified system for a concrete application. To build the system, we significantly extend our previous work by incorporating a crowdsourcing component to facilitate further uncertainty handling and a component for traffic modelling. The first component queries volunteers close to the sensors that disagree and estimates what has actually happened given the participants' reliability. The benefits of this approach are two-fold. First, more accurate information is directly given to end users. Second, the event processing component of our system makes use of the crowdsourced information to minimise the use of unreliable sources. The traffic modelling component may also use the crowdsourced information to resolve data sparsity.

We illustrate our approach using large, heterogeneous data streams concerning urban traffic management in the city of Dublin. We describe the requirements that come up including data sources, analysis methods and technology, and visualisation. The data we use[2] come from the Sydney Coordinated Adaptive Traffic System (SCATS) sensors, i.e. fixed sensors deployed on intersections to measure traffic flow and density, and bus probe data stating, among others, the location of each bus as well as traffic congestions.

The remainder of this paper is organised as follows. Section 2 describes the architecture of our system. Then, Sections 3–6 present each of the main components. Section 7 presents our empirical evaluation, showing the method feasibility. Finally, Section 8 summarises our work.

2. SYSTEM ARCHITECTURE

The general architecture of our system for urban traffic management is given in Figure 1. In this section, we describe the input and output of the system, the individual components that perform the data analysis, and the stream processing connecting middleware.

Input: Two types of sensor are considered as event sources. Buses transmit information about their position and congestion and vehicle detectors of a SCATS system are installed at intersections and report on traffic flow and density. The raw

[2] `www.dublinked.ie`

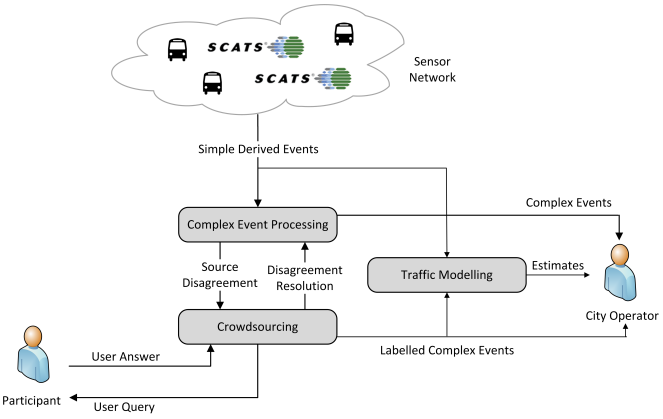

Figure 1: Overview of the system architecture.

input from these sensors is not directly processed though. Instead, mediators are involved that filter and aggregate the raw data. A lack of control over these pre-processing steps that are interwoven with the communication infrastructure, therefore, induces uncertainty for the low-level events that are actually processed by the system. This aspect is highlighted by the notion of a simple, derived event (SDE), which is the result of applying a computational derivation process to some other event, such as an event coming from a sensor [21]. A stream of such time-stamped SDEs is the primary input of our system.

Additionally, the system may solicit input from citizens using a connected crowdsourcing component. The output of crowdsourcing is fed to the computing components of the system, to improve the accuracy of the results.

Output: The system helps an operator manage the traffic situation, by integrating available traffic information from the different sources, which can then be used to issue alerts when issues that may impact traffic are identified. An important requirement is to have a simple, intuitive interactive map to present all traffic information and alerts.

Stream Processing Component: The backbone of our solution is a stream processing component, which couples the output from the sensors with further data analysis components. Stream processing is realized with the *Streams* framework [4]. It provides a language for the description of data flow graphs, which are then compiled into a computation graph for a stream processing engine.

Data Analysis Components: The system uses components for traffic modelling, complex event processing and crowdsourcing. Collectively, these components implement the monitoring logic of the system. The crowdsourcing component has two independent parts: the query modelling part whose objective is to select the humans that will be answering a question, and a query execution engine which deploys and executes the question.

Using Streams, SDEs are forwarded to a traffic modelling component that deals with data sparsity, i.e. makes congestion estimates in areas with low or non-existent sensor coverage. SDEs are also forwarded to a complex event processing engine that identifies complex events (CE) of interest. A CE is a collection of events that satisfies a certain specification comprising temporal and, possibly, atemporal constraints on its deriving events, either SDEs or other CEs. Identified

CEs may then be directly forwarded to end users (city operators) in order to gain insights on the current traffic situation. However, the aforementioned uncertainty stemming from the pre-processing of sensor readings may lead to situations that cannot be clearly identified. Instead, CEs that relate to inconsistencies in the event sources are detected. These CEs are then forwarded to the crowdsourcing component, which aims at reducing the uncertainty by human input. For a source disagreement event emitted by the complex event processing component, the crowdsourcing component selects one or more humans that act as system participants. By answering a specific question, they allow for resolving source disagreements. These results are used in two ways. On the one hand, they are fed into the complex event processing component and the traffic modelling component, thereby supporting adaptability of these components. On the other hand, CEs are labelled with the details obtained from the participants and forwarded to city operators, allowing for deeper insights on the traffic situation.

3. STREAM PROCESSING

The *Streams* framework [4] that is the backbone of our system provides a XML-based language for the description of data flow graphs that work on sequences of data items which are represented by sets of key-value pairs, i.e. event attributes and their values. The actual processing logic, i.e. the nodes of the data flow graph, is realised by processes that comprise a sequence of processors. Processes take a stream or a queue as input and processors, in turn, apply a function to the data items in a stream. All these concepts are implemented in Java, so that adding customized processors is realised by implementing the respective interfaces of the Streams API. In addition, Streams allows for the specification of services, i.e. sets of functions that are accessible throughout the stream processing application.

Using these concepts, our stream processing component includes the following parts:

- Input handling processes: all SDEs emitted by buses form one stream, while the SDE emitted by vehicle detectors of a SCATS system are referenced by four streams, one per region of Dublin city.
- Event processing processes: the definitions of complex events (CE)s are wrapped by specific processors that realise an embedding of the complex event processing component in the Streams environment.
- Crowdsourcing processes: the selection of participants from which feedback should be sought, the generation of the actual queries, and the processing of responses are also implemented by specific processors.
- Traffic modelling processes: the procedure for making congestion estimates at locations with low sensor coverage is wrapped as a Streams service.

For complex event processing, our solution relies on the Event Calculus for Run-Time reasoning (RTEC)[3] [2], a Prolog-based engine, which is detailed below. We integrated RTEC by a dedicated processor in Streams that would forward the received SDEs to an RTEC instance using a bidirectional Java-Prolog-Interface. Then, the actual event processing is triggered asynchronously and the derived CEs are emitted to a queue in the Streams framework.

Crowdsourcing essentially involves two steps, the generation of the queries and the processing of participant responses. In our solution, each of these steps is implemented by a dedicated processor. That is, upon the reception of a respective event (source disagreement) indicating that feedback should be sought, a first processor takes events as input and queries actual participants via an interface. Responses to these queries, in turn, represent an event stream. The responses are merged by a second processor to come up with an approximation of the probabilities for the different possible answers. This second processor also estimates participant reliability.

[3] `http://users.iit.demokritos.gr/~a.artikis/EC.html`

4. COMPLEX EVENT PROCESSING

Our CE recognition component is based on the Event Calculus for Run-Time reasoning (RTEC) [2]. The Event Calculus [15] is a logic programming language for representing and reasoning about events and their effects. The benefits of a logic programming approach to CE recognition are well-documented: such an approach has a formal, declarative semantics, and direct routes to machine learning for constructing and refining CE definitions in an automated way. The use of the Event Calculus has additional advantages: the process of CE definition development is considerably facilitated, as the Event Calculus includes built-in rules for complex temporal representation and reasoning, including the formalisation of inertia. With the use of the Event Calculus, one may develop intuitive, succinct CE definitions, facilitating the interaction between CE definition developer and domain expert, and allowing for code maintenance.

To make the paper self-contained, we summarise the essentials of the CE recognition model based on [2, 3]. We adopt the common logic programming convention that variables start with upper-case letters and are universally quantified, while predicates and constants start with lower-case letters.

4.1 Representation

In RTEC, event types are represented as n-ary predicates *event(Attribute1,...,AttributeN)*, such that the parameters define the attribute values of an event instance *event(value1,...,valueN)*. An example from the Dublin traffic management scenario is the type of SDE emitted by buses, $move(Bus, Line, Operator, Delay)$, which states that *Bus* is running in *Line* with a *Delay*, and operated by *Operator*. Thus, a specific event instance is an instantiation of this predicate, e.g. *move(33009, r10, o7, 400)*.

Time is assumed to be linear and discrete, represented by integer time-points. The occurrence of an event E at time T is modelled by the predicate $\mathsf{happensAt}(E, T)$. The effects of events are expressed by means of *fluents*, i.e. properties that may have different values at different points in time. The term $F = V$ denotes that fluent F has value V. $\mathsf{holdsAt}(F = V, T)$ represents that fluent F has value V at a particular time-point T. Interval-based semantics are obtained with the predicate $\mathsf{holdsFor}(F = V, I)$, where I is a list of maximal intervals for which fluent F has value V continuously. holdsAt and holdsFor are defined in such a way that, for any fluent F, $\mathsf{holdsAt}(F = V, T)$ iff time-point T belongs to one of the maximal intervals of I for which $\mathsf{holdsFor}(F = V, I)$. Table 1 presents the main RTEC predicates.

Fluents are *simple* or *statically determined*. For a simple fluent F, $F = V$ holds at time-point T if $F = V$ has been *initiated* by an event at some time-point earlier than T (using predicate initiatedAt), and has not been *terminated* in the meantime (using predicate terminatedAt), which implements

Table 1: Main predicates of RTEC.

Predicate	Meaning
happensAt(E,T)	Event E occurs at time T
holdsAt$(F=V,T)$	The value of fluent F is V at time T
holdsFor$(F=V,I)$	I is the list of the maximal intervals for which $F=V$ holds continuously
initiatedAt$(F=V,T)$	At time T a period of time for which $F=V$ is initiated
terminatedAt$(F=V,T)$	At time T a period of time for which $F=V$ is terminated
relative_complement_all(I',L,I)	I is the list of maximal intervals produced by the relative complement of the list of maximal intervals I' with respect to every list of maximal intervals of list L
union_all(L,I)	I is the list of maximal intervals produced by the union of the lists of maximal intervals of list L
intersect_all(L,I)	I is the list of maximal intervals produced by the intersection of the lists of maximal intervals of list L

the *law of inertia*. Statically determined fluents are defined using interval manipulation constructs, such as union_all, intersect_all and relative_complement_all (cf., Table 1).

The input SDE streams are represented by logical facts that define event instances, with the use of the happensAt predicate, or the values of fluents, with the use of the holdsAt predicate. Taking up the earlier example, facts of the following structure model the bus data stream:

$$\begin{aligned}&\mathsf{happensAt}(move(Bus,Line,Operator,Delay),\ T)\\&\mathsf{holdsAt}(gps(Bus,Lon,Lat,Direction,Congestion)=\mathsf{true},\ T)\end{aligned} \tag{1}$$

$gps(Bus, Lon, Lat, Direction, Congestion)$ states the location (Lon, Lat) of the Bus, as well as its direction (0 or 1) on the $Line$. Further, the gps fluent provides information about congestion (0 or 1) in the given location.

CEs, in turn, are modelled as logical rules defining event instances, with the use of happensAt, the effects of events, with the use of initiatedAt and terminatedAt, or the values of the fluents, with the use of holdsFor. For illustration, consider an instantaneous CE that expresses a sharp increase in the delay of a bus:

$$\begin{aligned}&\mathsf{happensAt}(delayIncrease(Bus,Lon',Lat',Lon,Lat),\ T)\leftarrow\\&\quad\mathsf{happensAt}(move(Bus,\_,\_,Delay'),\ T'),\\&\quad\mathsf{holdsAt}(gps(Bus,Lon',Lat',\_,\_)=\mathsf{true},\ T'),\\&\quad\mathsf{happensAt}(move(Bus,\_,\_,Delay),\ T),\\&\quad\mathsf{holdsAt}(gps(Bus,Lon,Lat,\_,\_)=\mathsf{true},\ T),\\&\quad Delay-Delay'>d,\\&\quad 0<T-T'<t\end{aligned}$$

'_' is a 'free' Prolog variable that is not bound in a rule. A $delayIncrease(Bus, Lon', Lat', Lon, Lat)$ CE is recognised when the delay value of a Bus increases by more than d seconds in two SDEs emitted in less than t seconds. A CE of this type may indicate a congestion in-the-make between (Lon', Lat') and (Lon, Lat). This indication may be reinforced by instances of this CE type concerning other buses operating in the same area.

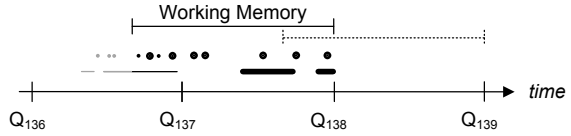

Figure 2: Event recognition in RTEC.

4.2 Reasoning

CE recognition is performed as follows. RTEC computes and stores the maximal intervals of fluents and the time-points in which events occur at specified query times $Q_1, Q_2, \ldots$. At each query time Q_i, only the SDEs that fall within a specified interval—the 'working memory' (WM) or 'window'—are taken into consideration: all SDEs that took place before or on $Q_i - WM$ are discarded. This way, the cost of CE recognition depends only on the size of WM and not on the complete SDE history. As a consequence, 'windowing' will potentially change the answer to some queries. Some of the stored sub-computations may have to be checked and possibly recomputed. Much of the detail of the RTEC algorithms is concerned with this requirement.

The size of WM, and the temporal distance between two consecutive query times — the 'step' ($Q_i - Q_{i-1}$) — are tuning parameters that can be either chosen by the end user or optimized for performance. In the common case that SDEs arrive at RTEC with delays, it is preferable to make WM longer than the step. This way, it becomes possible to compute, at Q_i, the effects of SDEs that took place in $(Q_i - WM, Q_{i-1}]$, but arrived after Q_{i-1}. This is illustrated in Figure 2. The figure displays the occurrences of SDEs as dots and a Boolean fluent as line segments. For event recognition at Q_{138}, only the events marked in black are considered, whereas the greyed out events are neglected. Assume that all events marked in bold arrived only after Q_{137}. Then, we observe that two SDEs were delayed i.e. they occurred before Q_{137}, but arrived only after Q_{137}. In our setting, the window is larger than the step. Hence, these events are not lost but considered as part of the recognition step at Q_{138}.

Note that increasing the WM size decreases recognition efficiency. This issue is illustrated in Section 7 where we evaluate empirically RTEC.

4.3 Event Recognition for Urban Traffic Management

The input to RTEC consists of SDEs that come from two heterogeneous data streams with different time granularity. First, buses transmit information about their position and congestions every 20-30 sec. The structures of the bus SDE is given by formalisation (1). Second, static sensors mounted on various junctions—SCATS sensors—transmit every 6 minutes information about traffic flow and density:

$$\mathsf{happensAt}(traffic(Int,A,S,D,F),\ T)$$

This instantaneous SDE expresses density D and traffic flow F measured by SCATS sensor S mounted on a lane with approach A into the intersection Int.

In collaboration with domain experts, several CEs have been defined over the input streams. These CEs relate to, among others, traffic congestion (in-the-make), and traffic flow and density trends for proactive decision-making. Traffic congestion is reported by SCATS sensors as well as buses.

The former is captured as follows:

$$\begin{aligned}
&\mathsf{initiatedAt}(\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{traffic}(\mathit{Int}, A, S, D, F),\ T), \\
&\quad D \geq \mathit{upper_Density_threshold}, \\
&\quad F \leq \mathit{lower_Flow_threshold} \\
&\mathsf{terminatedAt}(\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{traffic}(\mathit{Int}, A, S, D, F),\ T), \\
&\quad D < \mathit{upper_Density_threshold} \\
&\mathsf{terminatedAt}(\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{traffic}(\mathit{Int}, A, S, D, F),\ T), \\
&\quad F > \mathit{lower_Flow_threshold}
\end{aligned} \tag{2}$$

Here, *scatsCongestion* is a CE expressing congestion in a SCATS sensor and $\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true}$ is initiated when the density reported by SCATS sensor S, which is mounted on approach A of intersection *Int*, is above some threshold and traffic flow is below some other threshold (see the fundamental diagram of traffic flow[4]). Otherwise, $\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true}$ is terminated. The maximal intervals for which $\mathit{scatsCongestion}(\mathit{Int}, A, S) = \mathsf{true}$ holds continuously are computed by rule-set (2) and the domain-independent holdsFor predicate.

Given the above formalisation, we may define congestion with respect to a SCATS intersection, i.e. an intersection with at least one SCATS sensor. For example, we may define that a SCATS intersection is congested if at least n ($n > 1$) of its sensors are congested, or we may have a more structured intersection congestion definition that depends on approach congestion which in turn would depend on sensor congestion.

Congestion is also reported by buses—this is very useful as there are numerous areas in the city that do not have SCATS sensors. Consider the following formalisation:

$$\begin{aligned}
&\mathsf{initiatedAt}(\mathit{busCongestion}(\mathit{Lon}, \mathit{Lat}) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{move}(\mathit{Bus}, _, _, _),\ T), \\
&\quad \mathsf{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}_B, \mathit{Lat}_B, _, 1),\ T), \\
&\quad \mathit{close}(\mathit{Lon}_B, \mathit{Lat}_B, \mathit{Lon}, \mathit{Lat}) \\
&\mathsf{terminatedAt}(\mathit{busCongestion}(\mathit{Lon}, \mathit{Lat}) = \mathsf{true}, T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{move}(\mathit{Bus}, _, _, _),\ T), \\
&\quad \mathsf{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}_B, \mathit{Lat}_B, _, 0),\ T), \\
&\quad \mathit{close}(\mathit{Lon}_B, \mathit{Lat}_B, \mathit{Lon}, \mathit{Lat})
\end{aligned} \tag{3}$$

$(\mathit{Lon}, \mathit{Lat})$ are the coordinates of some area of interest, while $(\mathit{Lon}_B, \mathit{Lat}_B)$ are the current coordinates of a *Bus*. The *gps* fluent, like the *move* event, is given by the dataset. *close* is an atemporal predicate computing the distance between two points and comparing them against a threshold. $\mathit{busCongestion}(\mathit{Lon}, \mathit{Lat})$ starts being true when a bus moves close to the location $(\mathit{Lon}, \mathit{Lat})$ for which we are interested in detecting congestions, and (the bus) reports a congestion (represented by *1* in the *gps* fluent). Moreover, $\mathit{busCongestion}(\mathit{Lon}, \mathit{Lat})$ stops being true when a (possibly different) bus moves close to $(\mathit{Lon}, \mathit{Lat})$ and reports no congestion (represented by *0* in *gps*).

The two data sources, buses and SCATS sensors, do not always agree on congestion. Disagreement of the event sources is captured with the following formalisation:

$$\begin{aligned}
&\mathsf{holdsFor}(\mathit{sourceDisagreement}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ I) \leftarrow \\
&\quad \mathsf{holdsFor}(\mathit{busCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ I_1), \\
&\quad \mathsf{holdsFor}(\mathit{scatsIntCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ I_2), \\
&\quad \mathsf{relative_complement_all}(I_1,\ [I_2],\ I)
\end{aligned}$$

[4] http://en.wikipedia.org/wiki/Fundamental_diagram_of_traffic_flow

$\mathit{scatsIntCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int})$ is a CE expressing congestion in the SCATS intersection located at $(\mathit{Lon}_{Int}, \mathit{Lat}_{Int})$. relative_complement_all is an interval manipulation construct of RTEC (see Table 1). In $\mathsf{relative\_complement\_all}(I', L, I)$, I is the list of maximal intervals produced by the relative complement of the list of maximal intervals I' with respect to every list of maximal intervals of list L. The maximal intervals for which $\mathit{sourceDisagreement}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true}$ are computed only for the locations of SCATS intersections. A disagreement between the two data sources is said to take place as long as some buses report a congestion in the location $(\mathit{Lon}_{Int}, \mathit{Lat}_{Int})$ of a SCATS intersection, and according to the SCATS sensors of that intersection there is no congestion.

The detection of *sourceDisagreement* CE indicates veracity in the data sources. There are several ways to deal with this issue. Probabilistic event recognition techniques may be employed in order to deal with this type of uncertainty. Consider, for example, probabilistic graphical models [28], Markov Logic Networks [9, 26], probabilistic logic programming [25], and fuzzy set and possibility theory [19]. Although there is considerable work on optimising probabilistic reasoning techniques, the imposed overhead in the presence of large data streams, such as those of Dublin, does not allow for real-time event recognition [1].

In [3], we used variety of input data to handle veracity. The events detected on the bus data stream were matched against the events detected on the SCATS stream to identify mismatches that indicate uncertainty regarding the data sources. Temporal regions of uncertainty were identified from which the system autonomously decided to adapt its sources in order to deal with uncertainty, without compromising efficiency. More precisely, we assumed that SCATS sensors are more trustworthy than buses and used these sensors to evaluate the information offered by buses. A bus was considered unreliable when it disagreed with a SCATS sensor on congestion, and remained unreliable as long as it did not agree during its operation with some other SCATS sensor. The congestion information offered by unreliable buses, whether close to a SCATS sensor or not, was discarded.

In this paper, instead, we rely on crowdsourcing techniques to resolve unreliability in data sources. These techniques are presented in the following section. The benefits of this approach are two-fold. First, more accurate information is directly given to city operators in the case of source disagreement. Second, RTEC takes advantage of the crowdsourced information to minimise the use of unreliable sources. The rules below illustrate how this is achieved:

$$\begin{aligned}
&\mathsf{happensAt}(\mathit{disagree}(\mathit{Bus}, \mathit{Lon}_{Int}, \mathit{Lat}_{Int}, \mathit{positive}),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{move}(\mathit{Bus}, _, _, _),\ T), \\
&\quad \mathsf{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}_B, \mathit{Lat}_B, _, 1),\ T), \\
&\quad \mathit{close}(\mathit{Lon}_B, \mathit{Lat}_B, \mathit{Lon}_{Int}, \mathit{Lat}_{Int}), \\
&\quad \mathsf{not\ holdsAt}(\mathit{scatsIntCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ T) \\
&\mathsf{happensAt}(\mathit{disagree}(\mathit{Bus}, \mathit{Lon}_{Int}, \mathit{Lat}_{Int}, \mathit{negative}),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{move}(\mathit{Bus}, _, _, _),\ T), \\
&\quad \mathsf{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}_B, \mathit{Lat}_B, _, 0),\ T), \\
&\quad \mathit{close}(\mathit{Lon}_B, \mathit{Lat}_B, \mathit{Lon}_{Int}, \mathit{Lat}_{Int}), \\
&\quad \mathsf{holdsAt}(\mathit{scatsIntCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ T) \\
&\mathsf{happensAt}(\mathit{agree}(\mathit{Bus}),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(\mathit{move}(\mathit{Bus}, _, _, _),\ T), \\
&\quad \mathsf{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}_B, \mathit{Lat}_B, _, 1),\ T), \\
&\quad \mathit{close}(\mathit{Lon}_B, \mathit{Lat}_B, \mathit{Lon}_{Int}, \mathit{Lat}_{Int}), \\
&\quad \mathsf{holdsAt}(\mathit{scatsIntCongestion}(\mathit{Lon}_{Int}, \mathit{Lat}_{Int}) = \mathsf{true},\ T)
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{happensAt}(agree(Bus),\ T) \leftarrow \\
&\quad \mathsf{happensAt}(move(Bus,\_,\_,\_),\ T), \\
&\quad \mathsf{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 0),\ T), \\
&\quad close(Lon_B, Lat_B, Lon_{Int}, Lat_{Int}), \\
&\quad \mathsf{not\ holdsAt}(scatsIntCongestion(Lon_{Int}, Lat_{Int}) = \mathsf{true},\ T)
\end{aligned}
$$

According to the first two rules above, an event $disagree(Bus, Lon_{Int}, Lat_{Int}, Val)$ takes place when Bus moves close to the location (Lon_{Int}, Lat_{Int}) of a SCATS intersection and disagrees on congestion with the SCATS sensors of that intersection. Val is *positive* if the Bus states that there is a congestion and *negative* otherwise. Similarly, according to the last two rules above, an event $agree(Bus)$ takes place when Bus moves close to the location (Lon_{Int}, Lat_{Int}) of a SCATS intersection and agrees on congestion with the sensors of that intersection.

A bus is considered unreliable/noisy when it disagrees on congestion with the SCATS sensors of an intersection *and* the information offered by the SCATS sensors is correct according to the crowdsourced information:

$$
\begin{aligned}
&\mathsf{initiatedAt}(noisy(Bus) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(disagree(Bus, Lon_{Int}, Lat_{Int}, BusVal),\ T), \\
&\quad \mathsf{happensAt}(crowd(Lon_{Int}, Lat_{Int}, CrowdVal),\ T'), \\
&\quad BusVal \neq CrowdVal, \\
&\quad 0 < T' - T < threshold \\
&\mathsf{terminatedAt}(noisy(Bus) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(agree(Bus),\ T) \\
&\mathsf{terminatedAt}(noisy(Bus) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(disagree(Bus, Lon_{Int}, Lat_{Int}, Val),\ T), \\
&\quad \mathsf{happensAt}(crowd(Lon_{Int}, Lat_{Int}, Val),\ T'), \\
&\quad 0 < T' - T < threshold
\end{aligned}
\tag{4}
$$

$crowd(Lon_{Int}, Lat_{Int}, Val)$ is an event produced by the crowdsourcing component (details are given in Section 5). It states whether there was a congestion at the SCATS intersection located at (Lon_{Int}, Lat_{Int}) according to the human crowd. Val is *positive* if there was a congestion and *negative* otherwise. $noisy(Bus) = \mathsf{true}$ is initiated when a Bus disagrees on congestion both with the SCATS sensors of some intersection and the crowdsourced information. The last condition of the initiating rule requires that the crowdsourced information is used for evaluating the reliability of a bus only if it arrives within a specified period from the time of the source disagreement. $noisy(Bus) = \mathsf{true}$ is terminated when the Bus agrees with the SCATS sensors of some other intersection, or when it disagrees with SCATS sensors but the crowdsourced information proves the Bus correct.

An alternative definition of $noisy(Bus)$ is the following:

$$
\begin{aligned}
&\mathsf{initiatedAt}(noisy(Bus) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(disagree(Bus,\_,\_,\_),\ T) \\
&\mathsf{terminatedAt}(noisy(Bus) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(agree(Bus),\ T) \\
&\mathsf{terminatedAt}(noisy(Bus) = \mathsf{true},\ T') \leftarrow \\
&\quad \mathsf{happensAt}(disagree(Bus, Lon_{Int}, Lat_{Int}, Val),\ T), \\
&\quad \mathsf{happensAt}(crowd(Lon_{Int}, Lat_{Int}, Val),\ T'), \\
&\quad 0 < T' - T < threshold
\end{aligned}
\tag{5}
$$

According to the above rules, $noisy(Bus) = \mathsf{true}$ is initiated when a Bus disagrees on congestion with the SCATS sensors of some intersection, even when there is no crowdsourced information to identify the accurate data source. In other words, in the absence of information to the contrary, the SCATS sensors are considered more trustworthy than buses. $noisy(Bus) = \mathsf{true}$ is terminated, however, when there is crowdsourced information that proves the Bus correct. As before, $noisy(Bus) = \mathsf{true}$ is also terminated when there is source agreement.

Using $noisy(Bus)$, the *busCongestion* definition that reports congestion from bus data is adapted as follows:

$$
\begin{aligned}
&\mathsf{initiatedAt}(busCongestion(Lon, Lat) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(move(Bus,\_,\_,\_),\ T), \\
&\quad \mathsf{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 1),\ T), \\
&\quad \mathsf{not\ holdsAt}(noisy(Bus) = \mathsf{true}), \\
&\quad close(Lon_B, Lat_B, Lon, Lat) \\
&\mathsf{terminatedAt}(busCongestion(Lon, Lat) = \mathsf{true},\ T) \leftarrow \\
&\quad \mathsf{happensAt}(move(Bus,\_,\_,\_),\ T), \\
&\quad \mathsf{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 0),\ T), \\
&\quad \mathsf{not\ holdsAt}(noisy(Bus) = \mathsf{true}), \\
&\quad close(Lon_B, Lat_B, Lon, Lat)
\end{aligned}
\tag{3'}
$$

According to this new formalisation, the congestion information offered by a bus, whether close to a SCATS intersection or not, is discarded as long as the bus is considered unreliable, i.e. as long as the disagreements with SCATS sensors are resolved in favour of those sensors (when $noisy(Bus)$ is defined by rule-set (4)) or remain unresolved (when $noisy(Bus)$ is defined by rule-set (5)).

Given the crowdsourced information, we can also evaluate the reliability of SCATS sensors. The formalisation is similar and omitted to save space.

5. CROWDSOURCING

In this section we present the mechanisms we introduce to ameliorate the veracity problem of the data. Our main advance is the development of a novel crowdsourcing mechanism whose goal is to supplement the data sources through querying human volunteers, also called "participants", about the true state of the system. To minimise the impact on the participants, the crowdsourcing component is invoked by the complex event processing engine (RTEC) when a significant disagreement in the data sources is detected. Crowdsourcing relies on labels produced by imperfect experts—the participants—rather than on an oracle (e.g. a city employee). Crowdsourcing has enjoyed a recent rise in popularity due to the development of dedicated online tools, such as Amazon Mechanical Turk[5], and has been used for many complex tasks such as labelling galaxies [16], real-time worker selection [5] and solving various biological problems [13].

The main appeal of crowdsourcing is the reduced cost of label acquisition. Typically, the lower quality of the labels is compensated by acquiring several labels for each data item and combining them to produce a more accurate label. E.g. it has been known that the error of the average answer is usually smaller than the average error of each individual answer [12]. Developing increasingly better strategies to aggregate individual answers is an open research area. Many approaches try to model how reliable each participant is, and use participant reliability to improve the aggregation of answers. To this end, the Expectation-Maximization (EM) algorithm [23], Bayesian uncertainty scores [24] and sequential Bayesian estimation [10] have been used.

We present a crowdsourcing component that queries participants close to the location of a source disagreement event whenever requested by the CE processing component. The

[5] `www.mturk.com`

output of the crowdsourcing component is used for the resolution of the disagreement and sent to the end user/city operator, the CE processing component and the traffic modelling component.

In what follows, we describe our crowdsourcing model and briefly review the process of reliability estimation with the classical Expectation-Maximization (EM) algorithm [8, 22]. This algorithm needs to operate in batch mode, which is not acceptable for our large, streaming problem. Consequently, we then discuss an online version of the EM algorithm that supports online crowdsourcing task processing.

5.1 Crowdsourced Model

We model a source disagreement event as an unobserved categorical variable X_t, where $t \in \mathbb{N}$ is an index. Each variable X_t has a true value $x_t \in Val(X_t)$, where $Val(X_t)$ is the set of possible realizations or labels of X_t. Moreover, $X_t \perp X_{t'}\ \forall t \neq t'$, where $\perp$ denotes probabilistic independence. We assume that we have access to a prior distribution $P(X_t)$ over the possible values of the variable for every t. This distribution can either be provided by the CE processing component, or be the uniform distribution. E.g. if only 1 out of 4 buses at a given location indicates a congestion, the prior distribution could assign a lower prior probability to the congestion than if 3 out of 4 buses reported a congestion.

We denote by $y_{i,t}$ the answer given by participant i if he is queried about X_t, and $Y_{i,t}$ the associated variable. Moreover, we assume each participant i has a constant but unknown probability p_i to answer with a wrong label x_t when he is queried about an event X_t. When a participant does not give the true answer, he chooses another one at random. We also assume that $Val(Y_{i,t}) = Val(X_t)$, i.e. a participant queried about X_t is presented with all possible answers and none other. More formally,

$$P(Y_{i,t} = x_t | X_t = x_t) = 1 - p_i \qquad \forall i, t \tag{6}$$

$$P(Y_{i,t} = x | X_t = x_t) = \frac{p_i}{|Val(X_t)| - 1} \qquad \forall i, t, x \in Val(X_t) \backslash \{x_t\} \tag{7}$$

We also assume that $Y_{i,t} \perp Y_{i',t'}$ except if $t = t'$ and $i = i'$.

For each source disagreement event, we observe a set $\{Y_{i,t}\}_{i \in u_t}$ of answers, where u_t is the subset of participants queried based on the location. Our goal is to obtain the best prediction of $\hat{X}_t$.

Modifying the assumption on the parameterization of the conditional distribution of the answers or on the independence of the answers of different participants about the same event would not require big modifications to our approach. On the other hand, if other independence relationships no longer hold, the EM algorithm presented below may need to be altered significantly. E.g. consider two sensor disagreements X_t and $X_{t'}$ caused by the same bus during the same working memory. In our crowdsourcing model, we assume these two events to be independent. A more complex model could exploit a relationships between these events. This would require processing them together in the EM algorithm.

5.2 Estimation

If the parameters $\Theta \equiv \{p_i\}_i$ (the probability that each participant lies when queried) are known, inferring a posterior distribution $P(X_t | \{Y_{i,t}\}_{i \in u_t})$ is straightforward using Bayes rule. However, estimating these parameters is difficult. E.g. the maximum likelihood estimate $\hat{\Theta}$ of these parameters based on a crowdsourced data set of T unobserved events $X_{1:T} \equiv \{X_1, ..., X_T\}$ and associated answers $\mathcal{A}_{1:T} \equiv \{y_{i,t}\}_{i \in u_t, t \in 1:T}$ is the solution of the equation below:

$$\hat{\Theta} = \max_{\Theta} P(\mathcal{A}_{1:T} | \Theta) \tag{8}$$

$$= \max_{\Theta} E_{x_{1:T}} P(x_{1:T}, \mathcal{A}_{1:T} | \Theta) \tag{9}$$

Solving this equation is not analytically possible, because of the expectation on the hidden variables.

The Expectation-Maximization (EM) algorithm [8, 22] is a well-known method to solve this problem. It computes a sequence of parameters Θ_k that converges to a maximum. The algorithm alternates between computing an expectation of the likelihood, based on the observations and the current estimate of the value of the parameters, and maximizing this expectation to update the parameters:

$$Q_k(\Theta) = E_{x_{1:T} | \Theta_k, \mathcal{A}_{1:T}} \log P(x_{1:T}, \mathcal{A}_{1:T} | \Theta) \tag{10}$$

$$\Theta_{k+1} = \arg\max_{\Theta} Q_k(\Theta) \tag{11}$$

This algorithm operates in batch mode, which is problematic for stream processing. We could periodically evaluate the parameters Θ based on the full crowdsourced data set collected so far, but this would create scaling issues as this data set keeps growing. We could limit the number of events we work with to a manageable number, but such a strategy may induce the loss of all the answers provided by a participant. Indeed, we are only observing the answers of a (probably small) subset of participants for each event. Hence, if we operate on a subset of events, there's a risk we may discard all the answers of a participant.

Therefore, we use instead an online EM algorithm [6]. This algorithm can operate on one source disagreement event at the time, and both the event and the associated answers can be forgotten once this event has been processed. Discarding this information means that we cannot come back later and provide a more educated guess about the true value of the event. This is however only a minor drawback in our application. These events have a finite and short duration, so obtaining the true label is only relevant for a short time that depends on the working memory of the CE processing component. Moreover, and as opposed to many crowdsourcing applications, we can no longer ask questions about an event when it is over.

The online EM algorithm uses a stochastic approximation step to update the function $Q(\Theta)$ with a new event X_t rather than recomputing everything. Equation (10) of the EM algorithm therefore becomes:

$$\hat{Q}_t(\Theta) = (1 - \gamma_t) \hat{Q}_{t-1}(\Theta) + \gamma_t E_{x_t | \Theta_k, \mathcal{A}_t} \log P(x_t, \mathcal{A}_t | \Theta) \tag{12}$$

where the sequence $\gamma_1, \gamma_2, ...$ is such that $\lim_{T \to \infty} \sum_{t=1}^{T} \gamma_t = \infty$ and $\lim_{T \to \infty} \sum_{t=1}^{T} \gamma_t^2 < \infty$. As in the classical EM algorithm, Θ is then estimated by maximizing $\hat{Q}_t(\Theta)$.

In urban traffic management, we do not receive an answer from every participant for each source disagreement event. Therefore, we use a different stochastic approximation for every participant. In other words, we update each participant using a specific γ_{t_i}, where t_i is the number of times this participant has been queried so far. Applying this to the model described in Section 5.1, results in Algorithm 1. Every function $Q_k(\Theta)$ is a sum and each term corre-

Algorithm 1 Crowdsourcing

Require: $\{p_1, p_2, ...\}$ and $\{\gamma_1, \gamma_2, ...\}$
1: $t_i = 1 \quad \forall i$
2: **for all** $P(X_t), \mathcal{A}_t, Lon_t, Lat_t, T$ received **do**
3: **for all** $x \in Val(X_t)$ **do** {compute sufficient statistics}
4: $\hat{\alpha}(x) = P(X_t = x) \prod_{i \in u_t} P(Y_{i,y} = y_{i,t} | X_{i,t} = x)$
5: **end for**
6: **for all** $x \in Val(x)$ **do**
7: $\alpha(x) = \frac{\hat{\alpha}(y_{i,t})}{\sum_{x \in Val(X_t)} \hat{\alpha}(x)}$
8: **end for**
9: $Val = (\text{“}Traffic\ congestion\text{”} == \arg\max_x \alpha(x))$
10: send happensAt$(crowd(Lon_t, Lat_t, Val), T)$
11: **for all** $i \in u_t$ **do** {update parameters}
12: $p_i = (1 - \gamma_{t_i}) p_i + \gamma_{t_i} \left(1 - \frac{\alpha(y_{i,t})}{\sum_{x \in Val(X_t)} \alpha(x)}\right)$
13: $t_i = t_i + 1$
14: **end for**
15: **end for**

sponds to one source disagreement event. The parameters maximizing $Q_k(\Theta)$ will also be a sum where each term corresponds to one event and depends on the posterior probability $\alpha(x) \equiv P(X_t = x | \mathcal{A}_t, \{p_1, p_2, ...\})$ of the event. Algorithm 1 first computes these terms (lines 3 to 8), and then performs the stochastic approximation update of the parameter estimates (lines 11 to 14).

At line 10, the posterior distribution on the labels of the event is used to generate a message to the CE processing component, the traffic modelling component and/or the city operators. More precisely, we inform the interested parties whether the most likely label is a congestion or not.

5.3 Query Execution Engine

Having defined the query model, the next step is to employ a crowdsourcing query execution engine to communicate the queries to the participants while dealing with the challenges of the mobile setting: real-time performance and reliability. The functions we pursue are: (i) the provision of a communication backbone without effort from the user to reach him, and (ii) adaptive mechanisms that achieve real-time and reliable communication.

To maximize parallelism, the crowdsourcing component employs the MapReduce programming model [7, 14] to communicate the queries to the selected participants and enable them to do local processing. MapReduce is a computational paradigm that allows processing parallelizable tasks across distributed nodes. The model requires that the computational process is decomposed into two steps, namely map and reduce, where the following functions are used:

$$map(key; value) \rightarrow [(key2; value2)]$$

$$reduce(key2; [value2]) \rightarrow [finalvalue]$$

Each map function processes a key/value pair and produces an intermediate key/value pair. The input of the map function has the form $(key, value)$ and the output is another pair $(key2, value2)$. Each map function can be executed in parallel on different nodes. Each reduce function is used to merge all the individual pairs with the same key to produce the final output. Hence, it computes the final output by processing the list of values with the same intermediate $key2$.

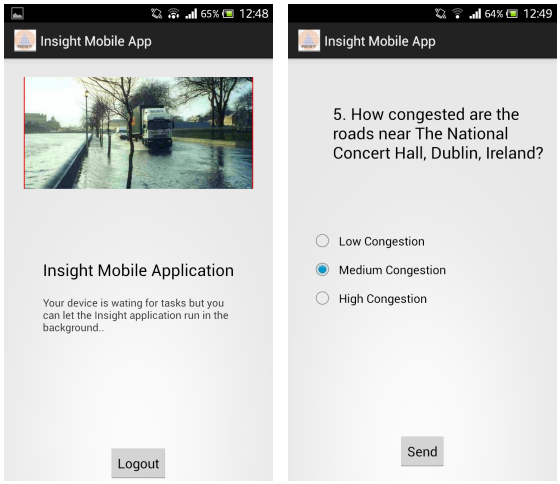

Figure 3: Crowdsourcing application.

In our system, the crowdsourcing query execution engine communicates the queries to workers—the participants—to answer specific questions about an event (map task), and aggregate the results (reduce task). The worker node receives the assigned task, processes the task and returns the answer, denoted as intermediate result, through the map function. After the crowdsourcing component collects all the answers of the subproblems (intermediate results), it combines them to form the output, which is the answer to the original query. This is achieved using the reduce function that is executed for all the intermediate results with the same key.

Each participant $i \in U$ registers with the query execution engine using a mobile device. The connection to the system requires the participant to: (1) connect to the Google Cloud Messaging (GCM) service to retrieve Push Notifications, and (2) connect to the Crowdsourcing Server using his id and identify himself as being a Map Worker. Then the participant can leave the application run on the background, where he can subscribe and retrieve tasks only when needed (see Figure 3). Note that the GCM service enables us to track the participant even if he changes his connection type (e.g. from WiFi to 3G), or when he remains behind a Network Address Translation-based routing device.

The query execution engine retrieves queries from the crowdsourcing component in the form of $query_q = \{Question_q, [answer_1, \ldots, answer_n]\}$, along with a list of Worker ids. In order to disseminate a $query_q$, the crowdsourcing component: (1) retrieves the registered online participants from the Crowdsourcing server, (2) selects the list of workers L_q to be queried based on the selected policy (e.g. location, reliability, etc), and (3) sends L_q and the $query_q$ to the Crowdsourcing Server and waits for the answers.

In case we have real-time response requirements for $query_q$, i.e. in the form of a time interval $deadline_q$, we should ensure that the time it takes to compute the query and communicate it to each selected participant should not exceed the $deadline_q$ requirement, i.e.:

$$comm_{iq} + comp_{iq} < deadline_q, \forall i \in L_q$$

Both the computation and the communication times can be estimated from historical data. The expected computation

time $comp_{iq}$ of each individual participant i to process a task q can be computed from the past executed tasks, and the communication time $comm_{iq}$ can be estimated from the communication time of the tasks executed previously in the participant's current location, since it depends on the network connection in that area—e.g. 2G or 3G.

The Crowdsourcing server disseminates the $query_q$ to the selected workers L_q by sending them a Push Notification that appears on their screen and notifies them by a vibration and a ringtone sound. Each worker can open the Map task by touching the notification, for which action the participant device connects with the Crowdsourcing server and retrieves the $query_q$. For instance, in traffic monitoring, the Map task is displayed on the participant's screen and he can select the answer (see Figure 3). After the Crowdsourcing Server has received answers from all Map workers or the reply time interval has expired, the Server selects a number of Reduce workers based on the selected policy. The Reduce workers retrieve the intermediate data, which are the answers of the Map workers, and aggregate them. Finally, the aggregated data are returned to the Crowdsourcing component. Although in the presented traffic monitoring example the computation is simple, we employ the MapReduce infrastructure to be able to additionally assign more complex queries. For instance, we could employ the sensors of the smartphones to extract data, such as their current speed or local humidity, as a Map task, and aggregate the intermediate data based on their density at the Reduce phase.

6. TRAFFIC MODELLING

In this section, we describe our approach to solve the data sparsity problem in our setting. Since data come from fixed installations (SCATS data) and specific routes (bus GPS-coded data), there are large parts of the city that are not covered. However, from a city monitoring view, it is important to offer the operator a current picture on the entire city area. We present a modelling technique that generalises the current observations to produce estimates for locations without sensors. A major requirement for such a technique is to be scalable to city-sized areas, and key to the scalability of our approach is focusing on modelling the usual, average case. The model is currently using SCATS data, and is trained using past data. The technique is designed to be general enough that any additional sources that can provide congestion information at specific locations can be incorporated in the training, including, specifically, the results of the crowdsourcing component.

The traffic network contains prior knowledge on movement through the city of Dublin. We model the edge oriented quantities within a Gaussian Process regression framework, similar to the approach in [18]. In the traffic graph $\mathcal{G}$ each junction corresponds to one vertex. To each vertex v_i in the graph, we introduce a latent variable f_i which represents the true traffic flow at v_i. The observed traffic flow values are conditioned on the latent function values with Gaussian noise ϵ_i

$$y_i = f_i + \epsilon_i, \epsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{13}$$

We assume that the random vector of all latent function values follows a Gaussian Process (GP), and in turn, any finite set of function values $\mathbf{f} = f_i : i = 1, \ldots, M$ has a multivariate Gaussian distribution with mean and covariances computed by the mean and covariance functions of the GP. The multivariate Gaussian prior distribution of the function values $\mathbf{f}$ is written as

$$P(\mathbf{f}|\mathbf{X}) = \mathcal{N}(0, \hat{K}) \tag{14}$$

where $\hat{K}$ is the so-called kernel and denotes the $M \times M$ covariance matrix; zero mean is assumed without loss of generality.

For traffic flow values at unmeasured locations u, the predictive distribution can be computed as follows. Based on the property of GP, the vector of observed traffic flows (v at locations $\overline{u}$) and unobserved traffic flows ($\mathbf{f}_u$ at locations u) follows a Gaussian distribution

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_u \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} \hat{K}_{\overline{u},\overline{u}} + \sigma^2 I & \hat{K}_{\overline{u},u} \\ \hat{K}_{u,\overline{u}} & \hat{K}_{u,u} \end{bmatrix}\right) \tag{15}$$

where $\hat{K}_{u,\overline{u}}$ is the corresponding entries of $\hat{K}$ between the unobserved vertices u and observed ones $\overline{u}$. $\hat{K}_{\overline{u},\overline{u}}$, $\hat{K}_{u,u}$, and $\hat{K}_{\overline{u},u}$ are defined equivalently. I is an identity matrix of size $|\overline{u}|$.

Finally the conditional distribution of the unobserved traffic flows are still Gaussian with the mean m and the covariance matrix Σ:

$$m = \hat{K}_{u,\overline{u}}(\hat{K}_{\overline{u},\overline{u}} + \sigma^2 I)^{-1}\, \mathbf{y}$$
$$\Sigma = \hat{K}_{u,u} - \hat{K}_{u,\overline{u}}(\hat{K}_{\overline{u},\overline{u}} + \sigma^2 I)^{-1}\, \hat{K}_{\overline{u},u}$$

Since the latent variables $\mathbf{f}$ are linked together in an graph $\mathcal{G}$, the covariances are closely related to the network structure: the variables are highly correlated if they are adjacent in $\mathcal{G}$, and vice versa. Therefore we can employ graph kernels [27] to denote the covariance functions $\hat{k}(x_i, x_j)$ among the locations x_i and x_j, and thus the covariance matrix $\hat{K}$.

The work in [18, 17] describes methods to incorporate knowledge on preferred routes in the kernel matrix. Lacking this information, we opt for the commonly used regularized Laplacian kernel function

$$\hat{K} = \left[\beta(L + I/\alpha^2)\right]^{-1} \tag{16}$$

where α and β are hyperparameters. L denotes the combinatorial Laplacian, which is computed as $L = D - A$, where A denotes the adjacency matrix of the graph $\mathcal{G}$. D is a diagonal matrix with entries $d_{i,i} = \sum_j A_{i,j}$.

7. EMPIRICAL EVALUATION

In this section we present the experimental evaluation of the main components of our system—complex event processing, crowdsourcing and traffic modelling. We used real data streams coming from the buses and SCATS sensors of Dublin city. The streams were collected between 1-31 January 2013 and comprise 13GB of data. The bus dataset includes 942 buses. Each operating bus emits SDEs every 20-30 seconds—on average, the bus dataset has a new SDE every 2 seconds. The SCATS dataset includes 966 sensors. SCATS sensors transmit information every six minutes. Both datasets are publicly available[2].

7.1 Complex Event Processing

We recognise CEs concerning traffic flow and density trends, traffic congestions and congestions in-the-make. Additionally, we compute the maximal intervals for which there is source disagreement, for resolution by means of crowdsourcing, and the intervals for which buses and SCATS sensors

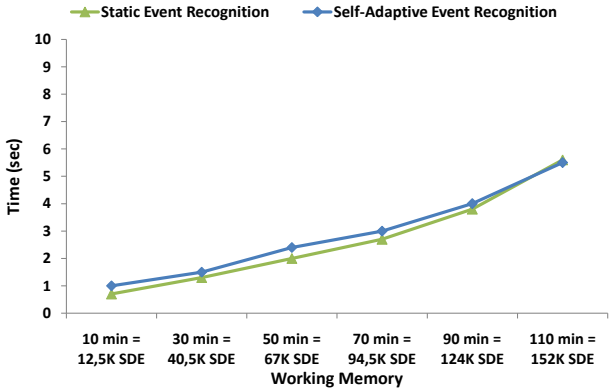

Figure 4: Event recognition performance.

are considered unreliable. The experiments were run on a computer with Intel i7 950@3.07GHz processors and 12GB RAM, running Ubuntu Linux 12.04 and YAP Prolog 6.2.2.

We present two sets of experiments. In the first, we performed 'static' recognition, that is, CE recognition that always takes into consideration all event sources. Then, we performed 'self-adaptive event recognition' where noisy sources are detected at run-time and the system discards them until they resume offering reliable information. CE recognition for traffic management, as defined here, is straightforward to distribute. E.g. in Dublin SCATS sensors are placed into the intersections of four geographical areas: central city, north city, west city and south city. We distributed CE recognition accordingly. We used four processors of the computer on which we performed the experiments—each processor computed CEs concerning the SCATS sensors of one of the four areas of Dublin as well as CE concerning the buses that go through that area. Figure 4 displays the average CE recognition times in CPU seconds. The working memory ranges from 10 min, including on average 12,500 SDEs, to 110 minutes, including 152,000 SDEs.

Figure 4 shows that self-adaptive CE recognition has a minimal overhead compared to static recognition. The overhead is due to computing and storing the maximal intervals of additional CEs, capturing the intervals for which some sources are considered unreliable. Figure 4 also shows that RTEC performs real-time CE recognition both in the static and the self-adaptive setting.

7.2 Crowdsourcing

Estimation. The crowdsourcing component was simulated to evaluate the performance of the online Expectation-Maximisation (EM) algorithm. We simulated 10 participants modelled as described in Section 5. We parameterized these participants using

$$\{p_i\}_{i=1}^{10} = \{0.05, 0.15, 0.2, 0.25, 0.25, 0.38, 0.4, 0.5, 0.75, 0.9\}$$

as their respective error probabilities. There are 4 possible answers. The first 7 participants are more likely to answer truthfully. The 8th participant has the same probability to give the true answer as one of the wrong ones. The 9th participant selects one of the 4 answers according to a uniform distribution. The last one is trying to mislead the system and is more likely to give a wrong answer than the 9th participant.

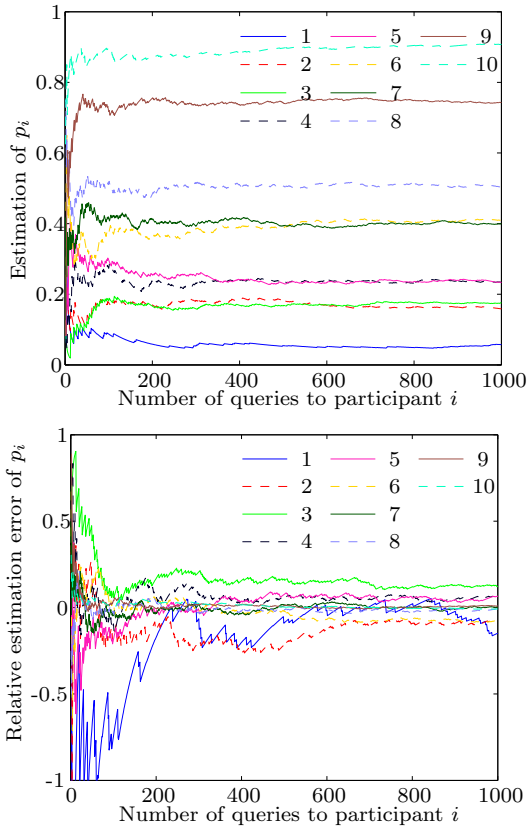

Figure 5: The estimation of the quality of each participant.

We used $\gamma_t = t/(t+1)$ for the stochastic approximation parameters. We initialize each p_i to 0.25, so we bias the initial parameters towards trustful participants. Using an unbiased initial estimate ($p_i = 0.75$) would prevent the parameters to be updated if the prior probability distributions $P(X_t)$ over the event labels were also uniform. All participants were queried about each sensor disagreement signalled by the CE processing component. Figure 5 illustrates the estimation of the quality of each participant (the probability that he provides a wrong answer when queried). The values of the estimation are displayed for each participant at the top and the relative estimation error at the bottom. Both estimations are functions of the number of calls to the crowdsourcing component.

A first observation is that the estimated values converge to the true value of the corresponding parameters. After processing approximately 100 calls, the ordering of the participant by quality is more or less correct, except for participants whose error probabilities are close (participants 2-3 and participants 6-7). Correctly estimating the quality of participants leads to a better assessment of the sensor disagreement, but it is also important for rewarding a participant. Indeed, a participant's quality may be a factor in the computation of the reward he receives for his contribution.

Most of the time (94% in this experiment) the posterior probability distribution is very peaked: the probability of one of the 4 explanations is greater than 0.99. Rarely, the answers provided by the participants are not sufficient to remove the uncertainty. E.g. the following posterior distribution $[\sim 0.49, \sim 0.41, \sim 0.09, < 0.01]$ does not provide a clear

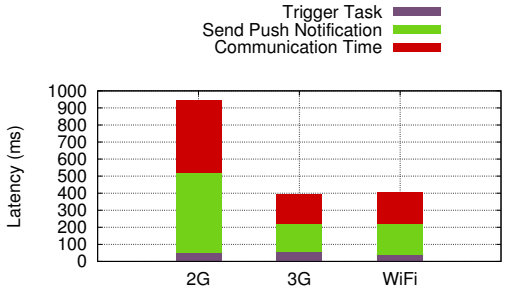

Figure 6: Crowdsourcing Query Execution Engine Latency.

explanation for the source disagreement. In general, however, crowdsourcing is able to resolve an overwhelming number of source disagreements.

Query Execution Engine. Figure 6 shows the latency of the individual steps of the crowdsourcing query execution engine using different connection types. The presented times are averages over 10 executions of crowdsourcing tasks for each connection type. We do not present the latency of the human responses, i.e. the latency to open the task and select an answer. We have observed that these times are typically a lot higher than the other steps. Figure 6 shows the latency to trigger a task, including the selection of the workers and the task assignment in the query execution engine, is minimal in all cases, since there is no communication with the participant devices, and ranges from 38 to 55 ms. On the other hand, the time needed to send a Push Notification from the participant device takes 467 ms on a 2G connection, while the 3G and WiFi connections only need 169 ms and 184 ms respectively. Note that a Push Notification requires from the query execution engine to send the notification to the Google Cloud Messaging server, and then this server forwards the notification to the device. Finally, Figure 6 shows the communication time that involves the communication to retrieve the task, once the task is selected, and send the answer back to the query execution engine. The 2G network experiences larger latency of 423 ms while the 3G network takes 171 ms and the WiFi connection 182 ms. Hence, although the end-to-end latency depends on the available network, even in case that only the 2G network is available the end-to-end latency would need less than a second to select a worker and communicate with him.

7.3 Traffic Modelling

For the traffic modelling experiments, the traffic network is generated using OpenStreetMap[6]—see Figure 7. In the pre-processing step, the network is restricted to a bounding window of the size of the city. Next, every street is split at every junction in order to retrieve street segments. Thus, we obtain a graph that represents the street network—see Figure 8. The SCATS locations, depicted as black dots in Figure 8, are mapped to their nearest neighbours within this street network. The sensor readings are aggregated within fixed time intervals. The hyperparametres are chosen in advance using grid search within the interval $[0, \ldots, 10]$. Using the pre-processed measurements, the Gaussian Process estimate is computed for the unobserved locations as described in Section 6. This step is repeated continuously. The results are plotted on a visual display—see Figure 9—and shaded according to their value. High values obtain a red colour while low values obtain green colour.

[6] www.openstreetmap.org

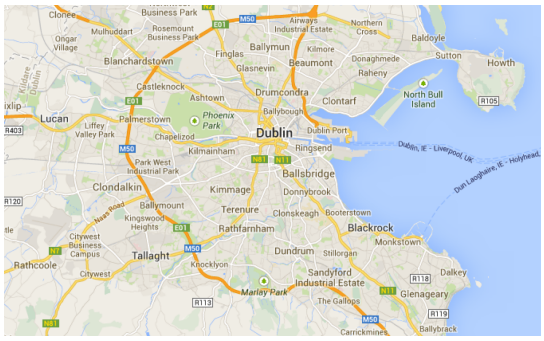

Figure 7: Map of Dublin, Ireland (from OpenStreetMap).

Figure 8: Street network and SCATS locations (black dots) in Dublin.

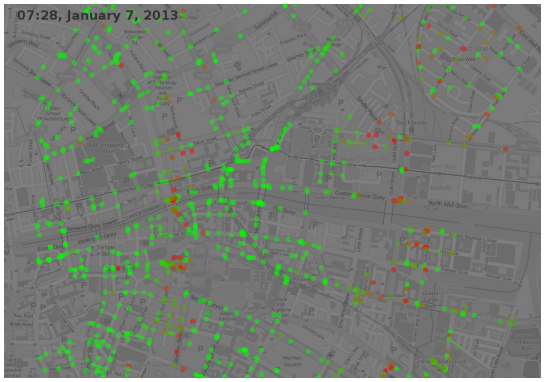

Figure 9: Traffic Flow estimates obtained by Gaussian Process Regression. Green dots correspond to low traffic whereas red dots indicate congested locations.

8. SUMMARY

We presented a system for heterogeneous stream processing and crowdsourcing supporting intelligent urban traffic management. Complex events related to traffic congestions (in-the-make) are detected from heterogeneous sources involving fixed sensors mounted on intersections and mobile sensors mounted on public transport vehicles. To deal with

the inherent data veracity, a crowdsourcing component handles and resolves source disagreement. Furthermore, to deal with data sparsity, a traffic modelling component makes congestion estimates in areas with low or non-existent sensor coverage. Our empirical evaluation on data streams from Dublin city showed the feasibility of the proposed system.

9. ACKNOWLEDGMENTS

This work is funded by the EU FP7 INSIGHT project (318225), the ERC IDEAS NGHCS project, and the Deutsche Forschungsgemeinschaft within the CRC SFB 876 "Providing Information by Resource-Constrained Data Analysis", projects A1 and C1.

10. REFERENCES

[1] A. Artikis, O. Etzion, Z. Feldman, and F. Fournier. Event processing under uncertainty. In *DEBS*, pages 32–43. ACM, 2012.

[2] A. Artikis, M. Sergot, and G. Paliouras. Run-time composite event recognition. In *DEBS*, pages 69–80. ACM, 2012.

[3] A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-adaptive event recognition for intelligent transport management. In *Big Data*. IEEE, 2013.

[4] C. Bockermann and H. Blom. The streams framework. Technical Report 5, TU Dortmund University, 12 2012.

[5] I. Boutsis and V. Kalogeraki. Crowdsourcing under real-time constraints. In *IPDPS*, pages 753–764, 2013.

[6] O. Cappé and E. Moulines. On-line expectation–maximization algorithm for latent data models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 71(3):593–613, 2009.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[8] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39:1–38, 1977.

[9] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool Publishers, 2009.

[10] P. Donmez, J. G. Carbonell, and J. G. Schneider. A probabilistic framework to learn from multiple annotators with time-varying accuracy. In *SDM*, 2010.

[11] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Company, 2010.

[12] F. Galton. Vox populi. *Nature*, 75:450–451, 1907.

[13] B. M. Good and A. I. Su. Crowdsourcing for bioinformatics. *Bioinformatics*, 2013.

[14] T. Kakantousis, I. Boutsis, V. Kalogeraki, D. Gunopulos, G. Gasparis, and A. Dou. Misco: A system for data analysis applications on networks of smartphones using mapreduce. In *MDM12*, 2012.

[15] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–96, 1986.

[16] K. Land, A. Slosar, C. Lintott, D. Andreescu, S. Bamford, P. Murray, R. Nichol, M. J. Raddick, K. Schawinski, A. Szalay, D. Thomas, and J. Vandenberg. Galaxy Zoo: the large-scale spin statistics of spiral galaxies in the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society*, 388:1686–1692, Aug. 2008.

[17] T. Liebig, Z. Xu, and M. May. Incorporating mobility patterns in pedestrian quantity estimation and sensor placement. In *Citizen in Sensor Networks*, volume LNCS 7685, pages 67–80. Springer, 2013.

[18] T. Liebig, Z. Xu, M. May, and S. Wrobel. Pedestrian quantity estimation with trajectory patterns. In *Machine Learning and Knowledge Discovery in Databases*, volume LNCS 7524, pages 629–643. Springer, 2012.

[19] H. Liu and H.-A. Jacobsen. Modeling uncertainties in publish/subscribe systems. In *ICDE*, 2004.

[20] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[21] D. Luckham and R. Schulte. Event processing glossary — version 1.1. Event Processing Technical Society, July 2008. `http://www.ep-ts.com/`.

[22] G. McLachlan and T. Krishnan. *The EM algorithm and extensions*, volume 382. John Wiley and Sons, 2008.

[23] V. C. Raykar, S. Yu, L. H. Zhao, G. H. Valadez, C. Florin, L. Bogoni, and L. Moy. Learning from crowds. *The Journal of Machine Learning Research*, 99:1297–1322, 2010.

[24] V. S. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *KDD*. ACM, 2008.

[25] A. Skarlatidis, A. Artikis, J. Filippou, and G. Paliouras. A probabilistic logic programming event calculus. *Theory and Practice of Logic Programming*, 2013.

[26] A. Skarlatidis, G. Paliouras, G. Vouros, and A. Artikis. Probabilistic event calculus based on markov logic networks. In *RuleML America*, pages 155–170, 2011.

[27] A. Smola and R. Kondor. Kernels and regularization on graphs. In *Proc. Conf. on Learning Theory and Kernel Machines*, pages 144–158, 2003.

[28] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Efficient processing of uncertain events in rule-based systems. *IEEE Trans. Knowl. Data Eng.*, 2011.