Automata-Based Dynamic Data Processing for Clouds

Cushing, R.; Belloum, A.; Bubak, M.; de Laat, C.

DOI
10.1007/978-3-319-14325-5_9

Publication date
2014

Document Version
Submitted manuscript

Published in
Euro-Par 2014: Parallel Processing Workshops

Automata-based Dynamic Data Processing for Clouds

Reginald Cushing[1], Adam Belloum[1], Marian Bubak[1,2], Cees de Laat[1]

[1] Informatics Institute, Universiteit van Amsterdam, Amsterdam, The Netherlands
[2] Department of Computer Science, AGH University of Science and Technology, Krakow, Poland
{R.S.Cushing, A.S.Z.Belloum, C.T.A.M.deLaat}@uva.nl
Bubak@agh.edu.pl

Abstract. Big Data is a challenge in many dimensions one of which is its processing. The often complicated and lengthy processing requires specialized programming paradigms to distribute and scale the computing power. Such paradigms often focus on ordering tasks to fit an underlying architecture. In this paper, we propose a new and complementary way of reasoning about data processing by describing complex data processing as set of state transitions, specifically, a non-deterministic finite automata which captures the essence of complex data processing. Through a P2P implementation of this model we demonstrate the dynamism, intrinsic scalability and adeptness to Cloud architectures.

Keywords: Automata-based modeling, Dynamic Cloud Computing, Data Defined Networking, Big Data, Distributed Data Processing, AaaS

1 Introduction

The growing emphasis on data processing capabilities due to the increasing volumes of data leads us to rethink the relationship between process and data [1]. Too often, the processing model and the data model are left as two disjoint areas; for example, workflow and MapReduce systems focus mainly on ordering tasks and do not intrinsically describe data transformations. Data often flows through these systems in a *pinball* fashion where it percolates through processes until it reaches the end process. Data provenance is the main approach to collecting data state as a means of its flow through the system but this is a *post-mortem* approach whereby we build a data flow graph after execution. Processes are often ordered to exploit the underlying infrastructure thus the same data processing workflow might look different for using grids, clouds or services. It is just to say that every data-centric workflow has an implicit data transition map which can be used to aid data processing, querying and data provenance. For example, coupling data with a state map in a workflow will provide a wider context for the data object as at any point in time you can deduce the current state of the data from the previous states and the possible future states i.e. what to expect from such data.

The proliferation of cloud-based processing means that data processing is increasingly becoming service oriented, specifically each single task is an Application-as-a-Service (AaaS). The potential scale of inter-cloud systems coupled with the dynamism of the infrastructure makes it increasingly difficult to coordinate a cohort of applications in a traditional central scientific workflow systems.

Asking “what do we want out of this data?” combined with the increased variety and velocity of data production means we need to be able treat data objects as distinct entities where each data object can take different processing paths inside a workflow just by changing states. For example, a typical parametric study workflow would setup a several process stages. The state of the data is usually the progress within the workflow itself which is not always indicative of the actual data state. Extracting features from data, the extracted features represent the new state and not so much the process that extracted them. A string that has been decided to be English by some function is said to be in an *English* state rather then in the state *outputOfFucntion*. This new level of information allows us to reason about data in terms of its state rather than just the processes.

In this paper we propose a data processing paradigm using Non-deterministic Finite Automata (NFA) where data transits from state to state as being processed. This enables us to construct data processing networks which capture the essence of data processing without knowledge of the underlying infrastructure. We prove this paradigm with an implementation, Pumpkin, which implements a data processing virtual network for cloud systems. The rest of the paper is structured as follows: Sec. 2 covers background material related to our contribution, Sec. 3 introduces the automata-based data process model and proof architecture description, Sec. 4 evaluates the applicability of the model to various use-case scenarios and finally, Sec. 5 discusses future work and concludes.

2 Related Work

In eScience, coordinating multiple tasks for running *in-silico* experiments is often the realm of Scientific Workflow Management Systems (SWMS). SWMS come in many forms and shapes and can vary in the computational model, types of processes used, and types of resources used. Many base their model on process flow [2] and also include a form of control flow [3], others implement data flow models [4] and some propose eccentric models such as based on chemical reactions [5]. A common denominator in most workflow systems is that the unit of reason is the process i.e. the abstract workflow describes a topology of tasks configured in a certain way. MapReduce [6] can be considered as a restricted workflow with two abstract tasks, *map()* and *reduce()*, aimed at exploiting data parallelism on locally distributed infrastructures such as data warehouses. In our approach we add a data layer on top of common workflow topologies. This approach allows us to model data processing as a sequence of state transitions and reason about data itself by reducing complex data processing to a set of atomic data transformations which can, inherently, be distributed.

Recently an initiative to provide a mechanism for content aggregation, called Research Object[3] has been proposed and is aiming at providing a way to annotate objects composing and a systems, making their type and relationships explicit.

Automata and state machines have been used in computing since its conception and have wide applications in most areas of computing. Of interest to us is its applicability to event stream processing [7] whereby automata is used to model queries in a distributed environment that queries streams of events such as stock events. The model used is an extension of the traditional NFA where instead of a finite input alphabet they read relational streams whereby state transitions are controlled by predicates. In our case predicates are dynamically generated by functions that process the data.

This work is inspired by the actor model proposed for concurrent computing, the actor model keep intern the mutable state and communicate using asynchronous messages [8]. Actors perform a certain task in response to a message, actors are autonomous and are able to create new actors creating a hierarchy of actors. In our approach, we build on top of the actor model whereby actors consume and produce state tags and do not send messages directly to each other but through a state tag routing system.

Coordination languages and more precisely the data flow-oriented category of coordination languages which focus on managing the communication among distributed collaborative applications as described in [9] and [10], are based on a similar approach where the coordination is not centralized in a specific component, but rather model as part of the network as in REO, an exogenous coordination model for constructing application using a calculus of channels and connectors [9]. Our approach is considerably simpler since it does not impose any constraints on the communication channels, the communication is always assumed to be asynchronous and the channels are not typed. We do not model communication *per se* but communication is a byproduct of a data state transition.

3 Automata-based Data Transformation Paradigm

We employ the formal definitions of NFA for describing data processing as an automaton. A NFA is defined as a 5 tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of state, Σ is the input alphabet, δ is the transition function, $q_0 \in Q$ is a start state, $F \subseteq Q$ is the set of final states. We extend the standard model so that we do not have a finite alphabet but a set of selection functions of the sort $\sigma(d, q)$ where q is the input state, $d \in D$ is the input data object to be processed. A selection function performs data processing, selects the next data state and returns a tuple (D', T), D' is a set of data objects produced by the processing of d and $T \subset Q$ is the set of new state tags generated non-deterministically by the selection function. A state tag is identical to a state but we distinguish between

[3] http://www.researchobject.org/ontologies/

a state identifier as a tag and the actual state as state. The state tag generated by the selection function controls the transition function. The transition and selection function can be considered as nested functions as in Fig. 1.

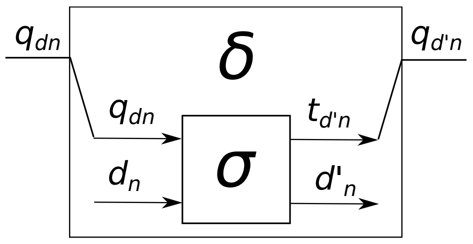

Fig. 1: Nested transition, δ, and selection, $\sigma \in \Sigma$, functions over $d_n \in D$. The transition is controlled by the output of the selection function.

In our model every data object has an associated NFA which describes the possible final state the data object it can transit to. The current state shows the progress of the data object. The transition function, $\delta(q, \sigma(d, q))$ defines the set of states that are reachable from state q with selection function σ. Given a data object at state q, only a subset of selection functions in Σ can act on the data and produce state transitions. We define this set of functions as $H_q \subseteq \Sigma$. The transition function can then be extended to multiple functions such that $\delta(q, H_q)$ defines all the states that are reachable from state q and functions H_q.

Figures 2 and 3 illustrate simple data automata. Figure 2 shows an automata which, when attached to a string data object, will filter the string as being English and having and *isa* and/or *hasa* relation. The initial state of the string is RAW since nothing is yet known about the string. $f(d)$ is the only function in H_{RAW} thus the transition $\delta(RAW, f(d, RAW))$ is taken. The functionality of $f(d, RAW)$ will determine the next state. The selection function $f(d, RAW)$ returns processed data and a state tag $ENG|NENG$ meaning the string is either English or not. If the string results in state ENG it can take further transitions. At state ENG, $(h(d), g(d)) \in H_{ENG}$ this means that both transitions $\delta(q_{ENG}, g(d, ENG))$ and $\delta(q_{ENG}, h(d, ENG))$ are taken simultaneously. This allows for string d to be concurrently filtered on multiple criteria.

Another NFA pattern is shown in Fig. 3. Here a data object is iteratively looped until a condition is met. The data object alternates between $READY$ and $UNSAT$ until a satisfiable condition such as a converged state is met at which point the data object transits to a SAT state.

3.1 Transformation Rules

Transformation rules are the set of production rules that define each transition function. These rules define mappings between state tags in the form:

- $(q_1, q_2, ...) \xrightarrow{A} (j_1, j_2, ...)$
- $(j_1, j_2, ...) \xrightarrow{B} (k_1, k_2, ...)$

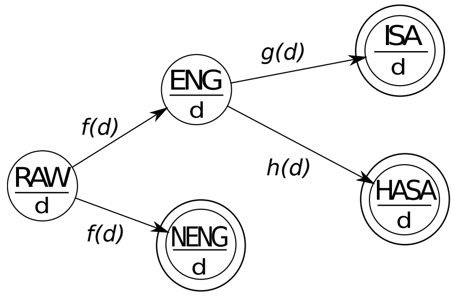

Fig. 2: A simple automaton showing the filtering of English language strings with an *isa* and *hasa* relation.

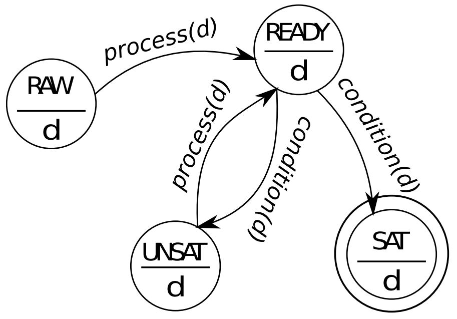

Fig. 3: An iterative pattern which processes data d until a final state SAT is reached.

Where q_n, j_n and k_n are members of Q and A, B are functions and members of σ. The global state tag graph is constructed from following state production rules. This allows for the state tag graph to grow dynamically by adding new rules to the database whereby the inference engine will automatically extend the mappings. As with other production rule based systems, querying the inference engine can reveal important information about possible processing of the data by forward chaining the rules. Thus, in a collaborative environment, one can discover new possibilities for data processing. On the other hand, using backward chaining, one can query the provenance of the data by showing what transitions a particular data object took to be in the current state. Another possibility is learning of possible gaps in a network where new functions are needed to fill the gaps.

3.2 Transition Operations

In our model selection functions are not simple one-to-one state tag transition. State tags can be combined using binary logic. This means that a selection function, $\sigma(d, q)$, can map an arbitrary number of state tags under these binary operations to other state tag statements. Figure 4 depicts some simple examples of these transition operations. The default transition operation is *or* thus if not explicitly states a function with multiple output state tags will either output one state tag or the other. Multiple input state tag means that a certain function accepts any data object in any of the defined states. The *and* operator acts as logic split and merge. A function simultaneously output multiple data objects with different state tags. Also, a function can accept **only** a number of data objects in a particular state thus multiple different transitions as needed to transform a data object into a new data object with different state. The latter is an example of correlating 2 datasets and producing a new, processed dataset.

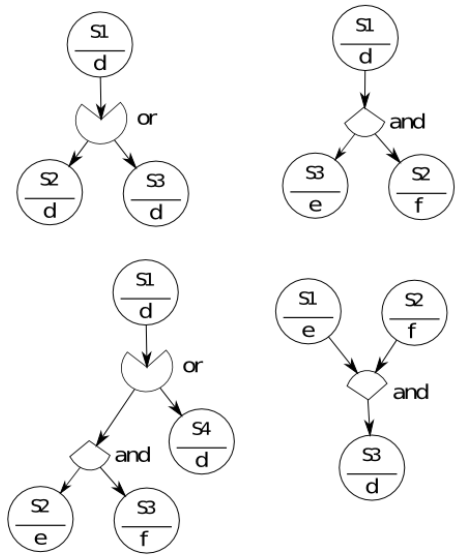

Fig. 4: Transition operations: Data in a state can transit to different simultaneous states. Two or more data objects can merge into a single state.

3.3 Architecture of Pumpkin Framework

The described model above is implemented as the Pumpkin framework[4]. The architecture implements a P2P distributed system for routing and processing data based on state tags. This overlay network between nodes is what we dub a Data Defined Network (DDF) as routes are setup based on the need for data transitions. The distributed characteristic of the system removes control centrality and the P2P characteristic allows for data being processed to pass directly between nodes thus minimizing third party data stores for intermediate data. The architecture is designed with the emerging cloud computing paradigm and virtual infrastructures in mind therefore one of the goals of Pumpkin is the coordination of bags of VMs to produce collective work.

The architecture of a single Pumpkin node in the network is illustrated in Fig. 5. The architecture builds around the concept of dynamically loading functions as is done in many web service containers. In addition to dynamically loading functions, a Pumpkin node implements a stack of control functionality to achieve the P2P capabilities. Most notable is the data routing based on state tags. Each Pumpkin node exposes a set of interfaces for accessibility. The interfaces can be categorized into control, data, and messages. Control and monitoring are user for user interaction and these include HTTP and FTP. Each node implements a TFTP server and client for file transfers between nodes. Messages in the form of data packets are the main mechanisms for communication between nodes.

Node discovery is done in two main ways. Nodes on the same local network will discover each other through UDP broadcasts. Pumpkin nodes on public facing interfaces can act as supernodes by disseminating peer information amongst

[4] https://github.com/recap/pumpkin

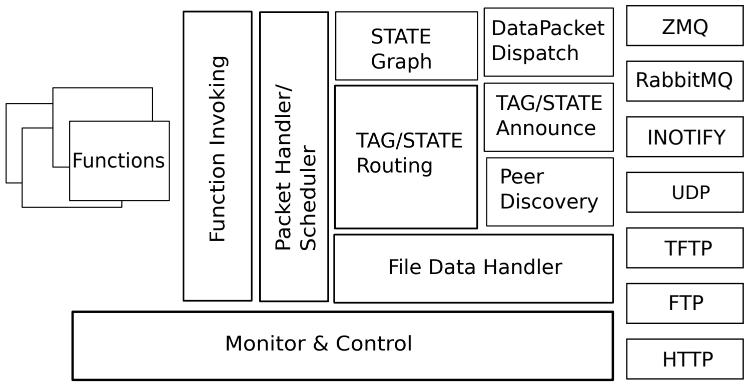

Fig. 5: Anatomy of a Pumpkin node. Connectivity is provided by a set of handlers (right), the core components (center) provide data packet handling and routing while the functions are the selection functions described in Sec. 3.

nodes. Through broadcasts each peer learns other peers' publish endpoints where information regarding state tags and functions is announced. This information is used by every node to build a view of the global state machine and a state tag routing table. The routing table is the crux of the Pumpkin network as it allows data state transitions over networks or shared memory for local functions. Pumpkin uses ZeroMQ[5] as its main messaging library. RabbitMQ[6] messaging is used as a proxy when Cloud instances are unable to open P2P connections.

Functions Functions implement the selection function in the NFA model described above. These are loaded dynamically at startup of every node or deployed during runtime. A function is essentially an implementation of an interface class that overrides a *run()*, and optionally *split(), merge(), on_load(), on_unload()* methods. Each function can accept multiple state tags and also produce multiple state tags. The numbers of possible transitions of a function is defined by the cross product between the input state tags and output tags.

A function is invoked as follows: upon reception of a data packet, Pumpkin extracts the relevant information from the packet and locate the appropriate class. After a staging sequence where data files might be downloaded from the previous node, the *run()* method is called. This step is bypassed if the received packet is part of a merge function in which case the *merge()* method is called on the packet. Each class can optionally define a *split()* and *merge()* methods. This scenario is useful when processing data packets in parallel would improve execution time. As an example, if a data packet containing a CSV file where each record is an input to a Monte-Carlo simulation then it would be advantageous to split the data before executing the *run()*. The split creates new fragments of the packet. The newly generated packets will be distributed to multiple instance of the same class. The results from the worker peers will be returned to the original peer where the *merge()* method is called on every packet thus the spliter node

[5] http://zeromq.org/
[6] http://www.rabbitmq.com/

acts as a temporary master node. The *merge()* method implements a data specific merging routine.

Data Packet Handling Communication between nodes is done with data packets. A data packet is a self-routable, encapsulation of a data object with header information to facilitate P2P state transitions described in the NFA model above. The architecture emphases state on the data packet meaning the data packet is stateful while the nodes are stateless to a certain extent. The data packet is modeled as a container into which arbitrary data can be placed, extracted, modified, and replaced. The container header describes the identification and processing status of the container. Identification is based on four fields; ship, container, box, and fragment. The analogy here is with cargo ships where a ship is a collection of containers and a container is a collection of boxes, i.e. a ship is the set of data D and a container is $d \in D$. A fragment ID is assigned when a box is split during processing into smaller chunks as would be the case to distribute data processing. The ship ID represents the execution context of the data; e.x. all data pertaining to an experiment would carry the same ship ID. Container and box ID are used to identify parts and sub-parts of the data. Together these fields represent the unique ID of data packet as it is being routed between nodes.

A mechanism for reliability can be enabled, albeit with a performance penalty. With reliability enabled nodes share responsibility of a data packet. The mechanism works as follows: Upon sending a data packet a node A retains the packet in a buffer until the peer node B sends back a Processed ACKnowledgment (PACK) back to A. A PACK is only sent after node B finished processing the packet and has dispatched it forward at which stage B becomes responsible for the packet. Upon reception of a PACK at A, the latter is relieved of its responsibility. If a PACK is not received in a timely fashion, node A will activate the TTL field in the data packet and resend the packet upon time expiration. Each node is equipped to detect duplicates, thus if B where to receive the duplicate packet it will reject it immediately. The TTL field can be tuned for different packets so that process hungry packets can have higher TTLs. As can be imagined this mechanism will put extra pressure on the system especially on packet buffers which are awaiting acknowledgments and thus the whole mechanism can be disabled in scenarios where it is not needed e.z. a streaming application.

4 Evaluation

As a proofing application we implemented a Tweeter filter engine based on the proposed model, and furthermore we evaluated the applicability to two real world scientific workflows.

Tweeter Filter Engine A Tweeter workflow is used as a proofing application for the distributed architecture. As Pumpkin is intended for virtual infrastructures and cloud computing it follows that testing is done on such an infrastructure. Our testbed is a private cloud infrastructure using OpenNebula on a 48-core AMD machine. Each VM is set to 0.1 CPU and 256 MB memory thus creating a relatively resource-lite VM. All VMs are configured to share a network bridge therefore all VMs appear to be on a LAN. Each VM is setup with a Pumpkin installation and deployed with different functions so as to create the test scenarios.

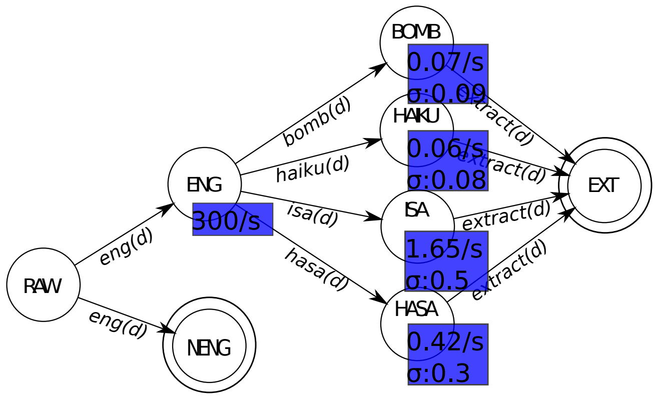

Fig. 6: A NFA describing data transitions. This NFA is mapped to 11 nodes where the functions *eng(), isa()* and *haiku()* where replicated on different VMs. The transition rates show the average rate tweets made a new transition; in this figure σ means standard deviation.

The Tweeter test scenario is setting up a state machine as a filter network for Tweeter feeds[7]. In this scenario every Tweet is packaged as a data packet and given the initial state of RAW. These packets are injected into the network and traverse the appropriate nodes to end in a final state. Each state transition acts as a filter thus tagging the data along the way with a state tag. The first filter *eng()* will tag the tweets with state ENG or state $NENG$. The ENG tagged data packets will move forward into the network while the other packets are dumped since a final state was reached. The last final state in Fig. 6 is an extraction state where we can monitor what packets are being generated. The results in Fig. 6 illustrates the rate at which data packets are transitioned into the 4 given filters i.e. accepted by the filter. Notable is that these state transitions are done in parallel due to the intrinsic parallelism defined in the state automaton in Fig. 6. Another level of parallelism is achieved from the replica functions hosted on different nodes. For this experiment there where *2* $\times$ *eng(), 2* $\times$ *isa(), 1* $\times$ *hasa(), 1* $\times$ *bomb(), 3* $\times$ *haiku()* functions hosted. The dynamism implemented in Pumpkin means that state transitions (in this case filters) can be added dynamically. Such an example is the function to state $BOMB$ where its VM startup was purposely retarded to demonstrate the dynamic addition of functions and the new flow of data. Along with these illustrated functions two

[7] http://snap.stanford.edu/data/twitter7.html

instances of the ENG transition function where running in parallel to each other and also in parallel to the other functions.

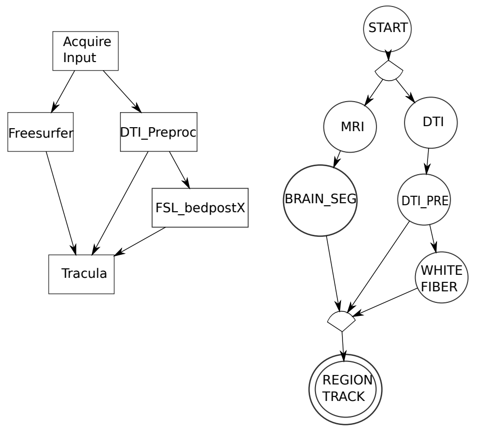

Fig. 7: Left: workflow for tracking brain fibers between regions. Right: Data transformations represented as a state tag graph.

Tracking Brain Regions Figure 7 shows a medical workflow for analyzing brain regions[8] using MRI and DTI. The characteristics of this workflow are that is it is a long running workflow (approximately 48 hours on commodity hardware) and it needs to run on multiple patients. In this case having a data processing structure represented as a data transition graph aids tracking patient processing data by knowing, at any time, in which state the data is. The main functions in the workflow (Left in Figure 7) are distributed on different virtual machines. The patient scans are then transformed using the data transformation graph (Right in Figure 7). Every patient's data will have an associated data transition graph while indicating the state and progress of the processing. These graphs are all processed on the same workflow. The architecture implicitly allows for scaling up by adding more virtual machines hosting certain functions of the workflow. This replication of tasks allows multiple patient data graphs to be processed simultaneously.

Blood Flow Simulation The blood flow simulation [11] and sensitivity analyses workflow is a computational intensive workflow with intricate data transition stages. The data transition graph aids in capturing the data flow as data enters different states. The characteristics of the workflow are that multiple models (in *Sepran* stage) can exist. The data transition graph can then easily capture data from different models since the output data would be in different states. The parallelism in this scenario is achieved two-fold; from distributing functions on different virtual machines and using split/merge functionality on seeds to

[8] http://www.bioinformaticslaboratory.nl/twiki/bin/view/EBioScience/TraculaDoc

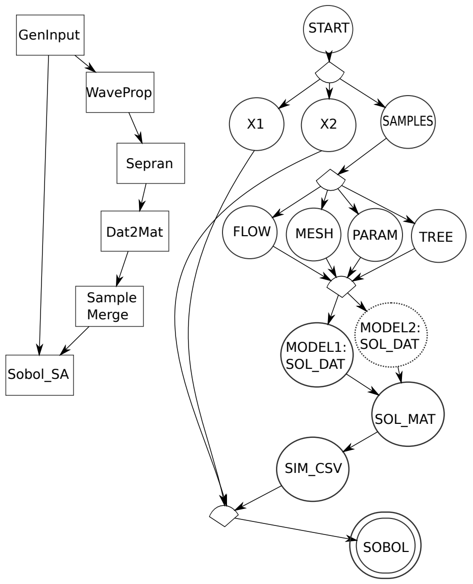

Fig. 8: Left: workflow for sensitivity analyses on blood flow simulations. Right: Data transformations represented as a state tag graph.

split the sample input into separate data packets which can then be processed simultaneously on different nodes hosting the same function.

5 Conclusion and Future Work

In this paper we described an innovative model for data processing based on NFA. The model takes a data centric approach to describe abstract data processing as a sequence of state transitions. The state of the data gives it context, we extended this context down to the processing and network levels where a distributed implementation, Pumpkin, exploits this information to process and route data to appropriate peer nodes for further processing. The results demonstrated how the model and the implementation work well in virtual environments and can be used to achieve distributed collaborative computing on cloud resources due to its P2P nature. We showed that the processing network can adapt quickly to new functionality through the discovery and dynamic loading. This means that the network can adapt quickly to varying data by injecting new functions. The P2P nature removes centralized bottlenecks in the network and therefore can scale well.

The usage of data packets as data processing parcels allows us to investigate added data routing attributes. The presented model data is routed based solely on its state. One can envision a data routing table with attributes such as energy and security thus a data packet can be routed and processed through greener or secure nodes.

In our implementation, data states are tags. These tags give limited context to the data while processing. Since tags can essentially be URLs to ontologies then we can think of tags as links to higher level semantics and therefore can explore this rich semantic level directly at processing and network layers. In Supercomputing 2013[9] we showed our initial approach to extending the data defined networking to Internet Factories [12] while ongoing research is looking into further symbioses between infrastructure and application such as congestion healing by self optimizing the infrastructure.

Acknowledgments: This work was partially funded by COMMIT (http://www.commit-nl.nl) and VPH-Share (http://www.vph-share.eu) projects.

References

1. Michael, K., Miller, K.W.: Big data: New opportunities and new challenges [guest editors' introduction]. Computer **46** (2013) 22–24
2. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, reloaded. In Gertz, M., Hey, T., Ludaescher, B., eds.: SSDBM 2010, Heidelberg, Germany (2010)
3. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on. (2004) 423–424
4. Cushing, R., Koulouzis, S., Belloum, A., Bubak, M.: Applying workflow as a service paradigm to application farming. Concurrency and Computation: Practice and Experience (2013) n/a–n/a
5. Fernandez, H., Tedeschi, C., Priol, T.: A chemistry-inspired workflow management system for scientific applications in clouds. In: eScience, IEEE Computer Society (2011) 39–46
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51** (2008) 107–113
7. Brenna, L., Gehrke, J., Hong, M., Johansen, D.: Distributed event stream processing with non-deterministic finite automata. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems. DEBS '09, New York, NY, USA, ACM (2009) 3:1–3:12
8. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410** (2009) 202 – 220 Distributed Computing Techniques.
9. Arbab, F.: Composition of interacting computations. In Goldin, D., Smolka, S., Wegner, P., eds.: Interactive Computation. Springer Berlin Heidelberg (2006) 277–321
10. Cortes, M.: A coordination language for building collaborative applications. Computer Supported Cooperative Work (CSCW) **9** (2000) 5–31
11. Huberts, W., et al.: A pulse wave propagation model to support decision-making in vascular access planning in the clinic. Medical engineering and physics **34** (2012)
12. Strijkers, R., Makkes, M.X., de Laat, C., Meijer, M.: Internet factories: Creating application-specific networks on-demand. To appear in Journal of Computer Networks (2014)

[9] http://youtu.be/DP5TLgW1hW4