Alleviating Hot-Spots in Peer-to-Peer Stream Processing Environments

Thomas Repantis and Vana Kalogeraki

Department of Computer Science & Engineering, University of California, Riverside, CA 92521
{trep,vana}@cs.ucr.edu

Abstract. Many emerging distributed applications require the processing of massive amounts of data in real-time. As a result, distributed stream processing systems have been introduced, offering a scalable and efficient means of in-network processing. Managing however the load among the nodes of such a large-scale, dynamic system in real-time is challenging. The peer-to-peer paradigm can help address these challenges via self-organization. We describe a self-managing architecture for identifying and alleviating hot-spots in a peer-to-peer stream processing environment. Resource monitoring and hot-spot detection are carried out by all peers independently, building upon a completely decentralized architecture. To alleviate hot-spots we empower peers to autonomously migrate the execution of stream processing components using a non-disruptive migration protocol. We have integrated our techniques in Synergy, our distributed stream processing middleware. The experimental evaluation of our implementation over PlanetLab demonstrates substantial performance benefits for distributed stream processing applications, with moderate monitoring and migration overheads.

1 Introduction

The need for real-time processing of high-volume, high-rate, continuous data streams has been documented in a variety of emerging Internet applications: i) Analysis of the clicks or textual input generated by the visitors of web sites such as e-commerce stores or search engines, to determine appropriate purchase suggestions or advertisements. ii) Monitoring of network traffic to detect patterns that proclaim attacks or intrusions, to filter out traffic that violates security policies, or to discover trends that can help with router configuration. iii) Customization of multimedia streams provided for entertainment or news coverage, to meet the interests, as well as the hardware and software capabilities of target users. iv) Processing of market data feeds for electronic trading, as well as surveillance of financial trades for fraud detection.

These types of applications have given rise to a new class of systems, called distributed stream processing systems [1–7]. Such systems attempt to provide Internet-scale stream processing engines, in which independent, reusable, and geographically distributed components process continuous data streams in real-time and are composed dynamically to generate outputs of interest. The distinct characteristic of distributed stream processing systems is that the data to be processed arrive in high rates, and often in bursts. Achieving low-latency processing of the streams under dynamically changing conditions, is a challenging task. In such an environment, the ability of the system to balance the load among hosts is a determinant factor of its success in coping with application demands in real-time. Most importantly, a distributed stream processing system needs to react to hot-spots caused by changes in data rates as well as due to the increasing number of new application requests. Traditional load balancing solutions only partially address the problem, since they do not directly address imbalances caused by fluctuations in the rates of long-running applications [8].

The unpredictability of the load, the large scale, and the distributed nature of a distributed stream processing system render its management a challenging task. Specifically, balancing the load among the stream processing nodes at run-time requires detailed monitoring of their utilization as well as complicated trade-offs that need to be evaluated frequently. Centralized supervision is impossible considering the data rates and the scale of the system. Hence, decentralized, self-organizing approaches seem more appropriate. Peer-to-peer computing can help in harnessing such complexity, by distributing the management decisions among all the nodes participating in the system.

We have been developing Synergy [9], a distributed stream processing middleware based on a peer-to-peer overlay, that dynamically balances the peers' load to satisfy the QoS demands of the applications. Synergy takes into consideration the QoS demands of the applications in the initial assignment [10], as well as in subsequent adaptations [11]. In this paper we describe our self-managing architecture for identifying and alleviating hot-spots in the face of dynamically changing workload and increasing number of applications in Synergy. To identify hot-spots and drive the load balancing decisions in a decentralized manner we implement our architecture on top of a Distributed Hash Table (DHT) peer-to-peer overlay [12]. To alleviate hot-spots we empower nodes to autonomously migrate the execution of stream processing components and we present a migration protocol that offers no disruption of the application execution and minimal performance impact. Our architecture enables nodes to accurately detect persistent overloads and load imbalances in order to issue a migration. To validate the performance of our techniques we have deployed our middleware on PlanetLab [13]. The evaluation of our prototype demonstrates substantial performance benefits for distributed stream processing applications, with moderate monitoring and migration overheads.

The rest of the paper progresses as follows: Section 2 describes the model of operation of a distributed stream processing system and the general architecture of Synergy, our middleware solution. Building upon this description, we describe our self-managing load balancing techniques in section 3, focusing on hot-spot identification and alleviation. The experimental evaluation is presented in section 4. Related work is discussed in section 5, while section 6 presents conclusions and avenues to future work.

2 The Synergy Distributed Stream Processing Middleware

In our Synergy distributed stream processing middleware [9] data streams arrive continuously from external sources (such as web users, monitoring devices, or a sensor network) into the stream processing system and need to be processed by stream processing components in real-time. Data streams consist of independent data tuples, which are produced continuously. They are processed by self-contained software components, that offer predefined operators. The operators can be as simple as a filter or a

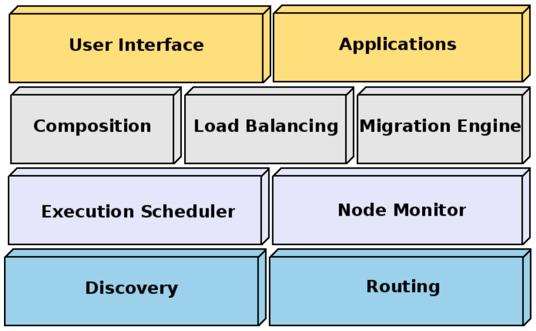

Fig. 1. The basic blocks of Synergy, our distributed stream processing middleware.

join, or as complex as transcoding or encryption. Operators are named based on a common ontology (e.g., $o_i.name = Encryption.AES.256bit$). Components are deployed in the distributed nodes (peers) according to their individual software capabilities or following criteria for the optimization of the performance of the whole system [5, 14–16].

The user executes a distributed stream processing application by submitting a request at one of the peers of the middleware, specifying the required operators and their dependencies. Then, the system runs a composition algorithm [9] to select the components on the peers to accomplish the application execution. The composition algorithm takes into account the application QoS requirements and resource availability and specifies the components that will constitute the application component graph. Once the components have been selected, each peer hosting one of these components is notified, so that the execution can begin. Once a peer receives such a notification, it invokes the particular component. Each component participating in an application execution is aware of its upstream components that supply it with data, and its downstream components, to which it sends processed data.

The peers of the middleware are connected via overlay links on top of the existing IP network. To facilitate decentralized component discovery, we organize the peers in a Distributed Hash Table (DHT), currently FreePastry [12]. Utilizing a DHT structure enables us to efficiently locate components and load information about the nodes. The basic blocks of our Synergy middleware running on each node are shown in Figure 1. The basic functionalities of a peer are the composition of applications that combine components at different nodes for their execution, and the engines that drive load management and migration decisions, based on monitoring of the current resource usage. These building blocks rely on message routing to other peers, and on component discovery, both of which are offered by the DHT infrastructure.

DHTs have been successfully used for decentralized discovery, enabling the mapping of keys to nodes, and routing query messages in logarithmic time [12]. Our extension to the traditional DHT model in which one component offering an operator would be stored in the node responsible for the operator's key, involves one more level of abstraction, that of a distributed inverted index. Following this approach, the peer responsible for an operator's key stores in a repository handlers to several peers that host components that actually offer the operator [3]. Thus, a peer of the DHT responsible for an operator holds handlers of all the peers that currently offer it in the system. These handlers are basically identifiers of the peers, enabling the routing of messages to them through the overlay. For example in Figure 3 peer B is responsible for keeping the handlers for the components that offer an aggregator operator. Therefore it keeps the handlers of peers B and C. While B happens to offer an aggregator component itself as well, this does not always have to be the case. While the peers have control over which components they host, they do not control which operator keys they are responsible for. This is determined by the DHT's protocol. Conversion of operator names to unique keys is done by applying a hash function (SHA-1). Configuration changes caused by node arrivals and departures are handled gracefully by the DHT. Whenever components are newly introduced or cease being offered, the peers hosting them register or unregister handlers for them on the DHT.

3 Load Management in a Peer-to-Peer Stream Processing Environment

We now describe our techniques for identifying and alleviating hot-spots at run-time in a self-managing way. We treat load imbalances in our peer-to-peer stream processing environment in two steps: First, we identify hot-spots in a decentralized fashion by utilizing the DHT infrastructure and we take load balancing decisions that will alleviate persistent overloads and load imbalances (section 3.1). Second, we resolve hot-spots using migration of components' execution (section 3.2).

3.1 Identifying Hot-Spots via a DHT

We identify hot-spots by monitoring the load of the stream processing peers and by comparing the load of peers that offer the same components. To achieve this in a distributed manner we build our load monitoring architecture on top of the DHT we use for component discovery.

Peers continuously monitor their load to enable load balancing decisions. The processor load is measured by parsing the `/proc` interface, where it is reported as the number of components in the run queue or waiting for disk I/O. Three load averages are maintained, namely the processor load during the last 1, 5, and 15 minutes.

Having each peer of the DHT maintaining handlers of all hosts offering a particular operator, as was described in section 2, facilitates load monitoring. Specifically, each peer responsible for an operator is also responsible for monitoring the load of the peers that host components offering it. It stores the handlers to the hosts offering the components in a peer-handler repository, and their corresponding loads in a load repository. We note that load monitoring peers are also hosting components. Thus, all responsibilities are shared among all nodes in a true peer-to-peer fashion. While we use the stored information to manage the stream processing load of the peers, we are not concerned with balancing the distribution of operator keys among peers, which is taken care of by the DHT protocol. Similarly, the DHT handles peer failures by transferring the responsibility for the operators for which a failed peer was responsible to other peers [12].

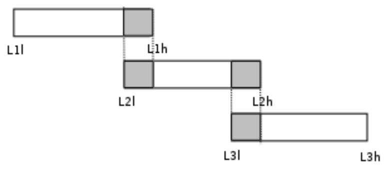

Fig. 2. Example of three overlapping load levels.

To avoid the communication overhead caused by polling, we enable the peers to inform the monitoring peers only when a significant change in their load occurs. Whenever a peer's load changes, it consults the DHT to determine the peers responsible for holding the handlers for all the components it offers. It then sends load update messages to them. For example, in Figure 3, peer B that offers a filter, an aggregator, and a transcoder, will send its load update messages to the responsible peers, C, B, and A respectively. One of the load update recipients happens to be peer B itself in this case. Propagating dynamic attributes, such as load measurements, efficiently, is a challenging task, as they may be need to be updated frequently. To reduce the communication overhead, our approach is to use k discrete levels of loads when describing the processing load of a peer. Each level is defined as an interval $L_i : [L_{il}, L_{ih})$, $L_{il} \leq L_{ih}$, within which the exact numeric load value is guaranteed to be. The basic idea is that the level eliminates all messages that are inside the interval. If the value of a load change l is small, *i.e.*, $l \in [L_{il} \leq L_{ih})$, then no messages need to be generated. Otherwise, load update messages need to be propagated. There is a tradeoff between the size of the interval and the number of messages generated. A narrow interval, *i.e.*, a small $|L_{ih} - L_{il}|$, enables more precise answers at the expense of a high communication cost, while a wide interval, *i.e.*, a large $|L_{ih} - L_{il}|$, reduces the communication cost but increases imprecision. To further reduce the number of messages propagated and also avoid cases where there is frequent change between two levels (*i.e.*, the load change is very small and falls in the boundaries of the levels, or there is load instability), we use overlapping levels; in which a small interval is common in two levels. Whenever the load changes but stays in the overlap region, no message needs to be generated. While load update messages are generated only when the load changes level, the actual load averages are then propagated instead of just the load level. Figure 2 graphically illustrates our overlapping levels approach. Load monitoring is performed by a different thread from the rest of the middleware functions such as stream processing. Thus, even if a peer has a high stream processing load, it can still send load update messages whenever needed. Load update messages can be propagated infrequently even if no load change occurred, to ensure that a peer has not failed. If a peer failure is detected, the components hosted by the failed peer need to be unregistered from the DHT and the applications that were using them need to recover using techniques such as the ones presented in [17].

We now describe the algorithm run by each peer to autonomously identify hot-spots. The goal of this algorithm is to identify persistently overloaded peers, for which lightly loaded peers with components offering the same operator exist. Since each peer that maintains handlers to all peers n_i hosting components of a particular operator o_j stores the current load values of the hosting peers n_i, the hot-spot identification algorithm is straightforward. Letting l_i be the load of each peer n_i offering an operator o_j a peer is responsible for on the DHT, we have the following algorithm for identifying hot-spots:

Criterion 1: Overload Detection. If $l_i > h$, where h is a threshold determining the sensitivity of our load balancing trigger, a hot-spot is identified. We consider balancing the load only of nodes that are identified as overloaded, instead of all nodes in pairs [8, 18], to avoid excessive migration overheads.

Criterion 2: Imbalance Detection. If $maxl_i - minl_i > l$, where l is a threshold determining the strictness of our load balancing goal, the load is not balanced equally. If the load was already balanced equally, a hot-spot could only be resolved by instantiating more components offering the particular operator, according to the current policy for placing them in the network, which is outside the scope of this paper [5, 14–16].

If $maxl_i - minl_i > l$, the identified hot-spot can be alleviated by load balancing. We explain how this is achieved via component migration from $\arg_i maxl_i$ to $\arg_i minl_i$, the peers hosting the components with the corresponding loads, in the next section (3.2).

To avoid identifying transient hot-spots, if the two criteria above are met, one additional check is performed. Namely, the average load during the last t minutes, in addition to l_i, which is the average load during the last 1 minute, is checked, to determine whether it is greater than h. Only then is an overload considered persistent and a migration is justified. Different values of t can be used to fine-tune the sensitivity of our component migration trigger. In our current implementation we have been monitoring the average load during $t = 1, 5, 15$ minutes.

The algorithm is run for every operator o_j a peer is responsible for, possibly identifying one or more migrations. The problem is complicated since different peers may concurrently identify the same hot-spot. For example, in Figure 3 an overload in peer C, which hosts an aggregator and a transcoder, may be detected by both peer A and peer B, which are responsible for detecting overloads in peers offering aggregators and transcoders respectively. To prevent an over-reaction to a hot-spot, in which more migrations than needed will take place, a peer that receives a migration decision while another migration is taking place ignores it. For example, if C receives a migration decision from A and later on a migration decision from B, it will only execute the first. More elaborate techniques, in which an overloaded peer would wait to see whether multiple migration decisions will arrive, and then decide which one to execute to more effectively lower its load, may also be possible. The fact that migration decisions are taken by peers autonomously and asynchronously renders the probability of concurrent migration decisions that apply to the same peer small. The probability is reduced further by the additional comparison of load averages over periods longer than 1 minute.

The three introduced threshold values h, l, t represent an important trade-off between being over- or under-sensitive to load imbalances. These tunable parameters also depend on the application and system characteristics. Our goal is not to find their optimal values, but we explore appropriate values in section 4.

3.2 Alleviating Hot-Spots via Migration

After the load balancing algorithm has reached a decision, the migration of component execution takes place to resolve a hot-spot. The execution of a particular com-

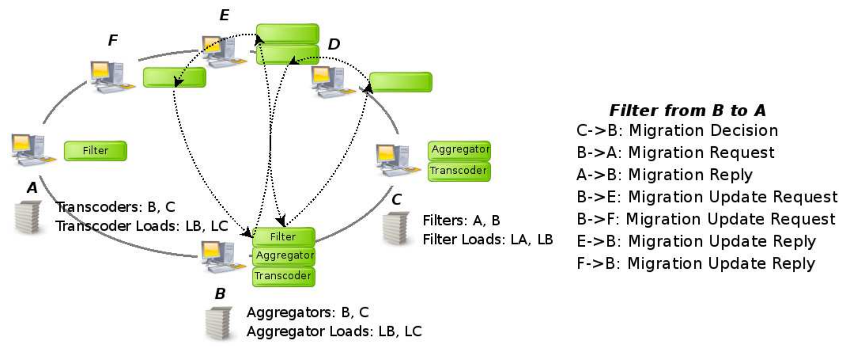

Fig. 3. Migration example.

ponent migrates to another peer that hosts the same component, redirecting the applications in which this component is participating. For example in Figure 3, the filter component of peer B is participating in two distributed applications, which will both be affected by the migration of it. While the component execution migrates, the component is not unregistered from the DHT, as the peer still has the capability of offering the same operator in the future. Our goal when choosing how to perform a migration is to offer no disruption in the application execution and minimal performance impact.

While some stream applications in which data units arrive in very high rates may actually afford missing some data tuples during the transition, we present a migration mechanism that also accommodates applications in which all data tuples must be delivered. The delivery of all data tuples in this case is checked using their sequence numbers, which indicate their requested delivery order.

It is important to perform the migration in the least intrusive way, to avoid affecting the performance of the applications as much as possible. Therefore we strive to improve upon the existing approaches of pausing the application execution or buffering incoming data tuples to be processed later [8, 18]. Thus, the connections from the new component to the upstream and downstream components are updated offline. In more detail, our non-disruptive migration protocol is as follows:

Phase 1. Once a peer decides the migration of a component's execution from a source to a target, it sends a *migration decision* message to the source. In Figure 3, C, responsible for balancing the load among peers offering the filter operator, decides that component execution needs to migrate from B to A. Hence, it sends a migration decision to B.

Phase 2. The source sends a *migration request* message to the target, in which the information for the corresponding component execution that need to be spawned is included. This basically includes which component's code will be executing, as well as which upstream and downstream components it will be communicating with. In Figure 3, B sends a migration request to A, informing it that it will be taking over the filter execution for the two distributed applications its filter component was currently participating in.

Phase 3. Once the new component has been instantiated, the target sends a *migration reply* message to the source, stating that it is ready to accept data tuples. In Figure 3, the migration reply is sent from A to B.

Phase 4. The source then sends *migration update request* messages to all upstream components of the migrating component, ordering them to update their downstream component to point to the one which resides on the target. In Figure 3, migration update requests are sent from B to E and to F.

Phase 5. The upstream components reply with a *migration update reply* message each, specifying that they have updated their downstream components. In addition they specify the sequence number of the last data tuple they sent before the update. They now send any further data tuples to the new component. In Figure 3, migration update replies are sent from E and F to B.

Phase 6. Once the source has collected replies from all upstream components and has processed the data tuples they specified as the last ones to be sent to it, it knows that the migration has completed. It then kills the component execution which has migrated.

Paying attention to the sequence numbers allows our protocol to know when a migration can complete, without disrupting the application execution. Furthermore, if a migrating component is stateful, its state must also be migrated once the last data tuple has been processed by its old host. Finally, updating the connections from the upstream components independently enables our protocol not to affect the application execution.

A common problem of migration protocols is the ping-pong phenomenon, manifesting itself as migrating processes or threads bouncing between two peers. In our case we have not observed this phenomenon often, as a migrating component execution will not return to its previous home, unless the relation of both the current and the average loads has been inverted.

4 Experimental Evaluation

To evaluate the performance of our hot-spot identification and alleviation mechanisms we integrated them in Synergy and performed experiments over the PlanetLab [13] wide-area network testbed. We used 30 hosts, each one of them issuing a request for a distributed stream processing application consisting of 5 operators. 10 types of unique operators were available in the system and they consisted of variations of DES encryption and decryption for different key sizes. Each peer was initially hosting 5 components that offered one of the 10 operators each. On every peer issuing a request a composition algorithm produced an application component graph consisting of 5 components. Subsequently, application execution began, with the sources producing data tuples 10 bytes large with random rates up to 80Kbps. Components processed data tuples as they arrived. The load monitoring thread on each host was running every minute, as was the load balancing thread. We used 5 load levels, overlapping by 0.2 as was described in section 3.1, while for the transitivity threshold t we used a value of 1 minute. We set the overload and imbalance thresholds, h, and l to 2.0 and 0.2 respectively. We also experimented with the sensitivity to all the tunable parameters.

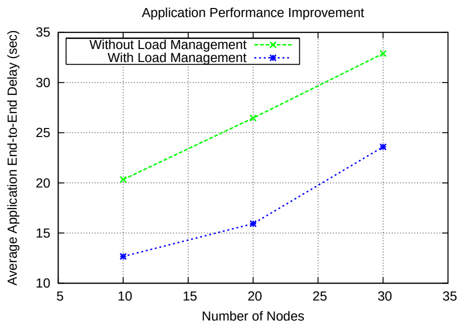

Fig. 4. Performance improvement.

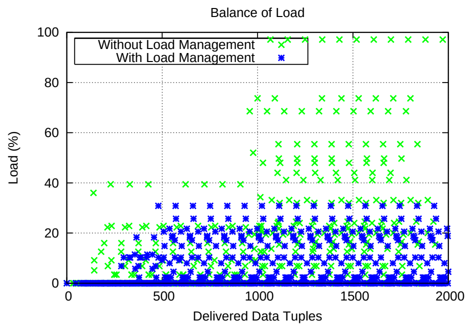

Fig. 5. Balance of load.

4.1 Application Performance Improvement

In the first set of experiments we investigated the benefit provided by our load management approach to the application performance, using the average end-to-end delay of the application as a metric. The end-to-end delay was defined as the time needed for a data tuple to arrive from the source to the receiver of the application, after being processed by all components of the application component graph. Figure 4 shows the significant reduction in the average end-to-end delay. The benefit is attributed to the fact that data tuples need to wait less time before they can be processed by the components.

4.2 Balance of Load

The load of the hosts as reported by the load update messages is shown in Figure 5. Every dot in the graph corresponds to a load value of a host at a particular point in time. The load values are plotted as a function of the data units that have been delivered to their final destinations, which is an indicator of the overall progress of the applications' execution. As more data units are inserted to the system, the load distribution disperses further when no load management is enabled.

4.3 Migration Protocol Overhead

We show the migration protocol overhead, defined as the number of messages required for the complete execution of the migration protocol, for the total number of migrations triggered, in Figure 6. We show the migration protocol overhead for different imbalance and overload thresholds, l and h respectively. Since the number of migrations is tunable by those thresholds, above which a migration is enabled, so is the protocol overhead. We note that even with very lax thresholds the average overhead is 249 migration-related messages for 2000 delivered data tuples. Including load update messages, which using 5 load levels we measured to be 259, we have a total of 508 overhead messages. If we take into account that every data tuple is transferred 4 times, from the source to the next stream processing component in the application component graph, and finally to the receiver, we have 8000 data message transfers, which gives a total average overhead of 508/8000 = 6.35%.

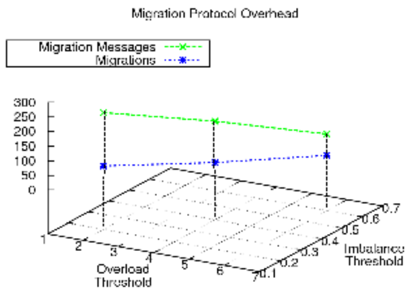

Fig. 6. Migration protocol overhead.

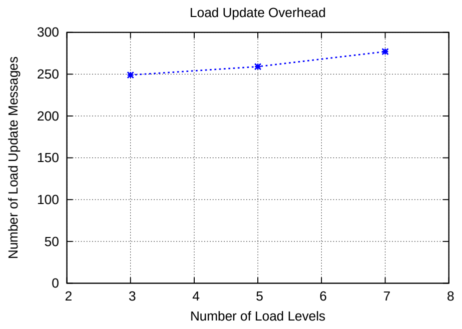

Fig. 7. Load update overhead.

4.4 Load Update Overhead

The overhead of the load monitoring mechanism is shown in Figure 7, as the number of load update messages propagated for different numbers of load levels. We always attribute the highest load level to any load above 10.0, which corresponds to a severe overload, and assign the rest of the load levels to loads from 0 to 10.0 accordingly. By varying the number of load levels from 3 to 7 we observed a moderate increase in the load update messages from 249 to 277. Thus, increasing the accuracy of our load monitoring mechanism only incurs a moderate increase in the load update overhead.

5 Related Work

Distributed stream processing systems have been the focus of a lot of recent research from different perspectives. Work on the placement of components to make efficient use of resources and to maximize application performance [5, 14, 15] is complementary to ours. Any technique for deploying new components can be used, once all the nodes hosting a particular component type are overloaded. Similarly, work on component composition [3, 9, 10] or application adaptation [11] can assist in load balancing, complementing our migration-based solution. Load balancing for distributed stream processing applications has also been studied [4, 8, 18–20]. We differ from these approaches in that we present a completely decentralized architecture that enables autonomous load balancing decisions. Furthermore, our focus is on load balancing rather than load shedding [21, 22], which can be employed when losing part of the application output is affordable. Load balancing in peer-to-peer systems has also been a topic of recent research [23, 24]. The focus there is on distributing objects evenly among peers, to improve resource utilization. In contrast, we focus on managing the load incurred by applications executing on top of a peer-to-peer network, in order to improve their performance. Finally, process migration has been employed to provide dynamic load distribution, fault resilience, improved system administration, and data access locality in a variety of domains, ranging from operating systems to batch application execution and mobile agents. An overview of related research can be found in [25]. Here we explore how migration can be used to alleviate imbalances caused by fluctuations in the rates of long-running distributed applications.

6 Conclusions

We have described a self-managing architecture for identifying and alleviating hot-spots in a peer-to-peer stream processing environment. Resource monitoring and hot-spot detection are carried out by all peers independently, building upon a totally decentralized architecture. To alleviate hot-spots we empower peers to autonomously migrate the execution of stream processing components and we have presented a migration protocol that offers no disruption in the application execution and minimal performance impact. We have integrated our techniques in Synergy [9], our distributed stream processing middleware. The experimental evaluation of our implementation over PlanetLab demonstrated substantial performance benefits for distributed stream processing applications, with moderate monitoring and migration overheads. In our future work we plan to extend our architecture to include multiple resource constraints, such as the peers' bandwidth, memory, or storage, in addition to the processor load.

References

1. Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings of the Second Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA. (2005)
2. Kumar, V., Cooper, B., Cai, Z., Eisenhauer, G., Schwan, K.: Resource-aware distributed stream management using dynamic overlays. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS, Columbus, OH, USA. (2005) 783–792
3. Gu, X., Yu, P., Nahrstedt, K.: Optimal component composition for scalable stream processing. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS, Columbus, OH, USA. (2005) 773–782
4. Gedik, B., Liu, L.: PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems, ICDCS, Providence, RI, USA. (2003) 490–499
5. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE, Atlanta, GA, USA. (2006) 49
6. Chandrasekaran, S., Cooper, O., Deshpande, A., andJ.M. Hellerstein, M.F., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA. (2003)
7. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA. (2003)
8. Xing, Y., Zdonik, S., Hwang, J.: Dynamic load distribution in the Borealis stream processor. In: Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan. (2005) 791–802

9. Repantis, T., Gu, X., Kalogeraki, V.: Synergy: Sharing-aware component composition for distributed stream processing systems. In: Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference, MIDDLEWARE, Melbourne, Australia. (2006) 322–341
10. Drougas, Y., Repantis, T., Kalogeraki, V.: Load balancing techniques for distributed stream processing applications in overlay environments. In: Proceedings of the 9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC, Gyeongju, Korea. (2006) 33–42
11. Chen, F., Repantis, T., Kalogeraki, V.: Coordinated media streaming and transcoding in peer-to-peer systems. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium, IPDPS, Denver, CO, USA. (2005) 56b
12. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, MIDDLEWARE, Heidelberg, Germany. (2001) 329–350
13. Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Peterson, L., Roscoe, T., Wawrzoniak, M.: Operating systems support for planetary-scale network services. In: Proceedings of the 1st Symposium on Networked Systems Design and Implementation, NSDI, San Francisco, CA, USA. (2004) 253–266
14. Seshadri, S., Kumar, V., Cooper, B.: Optimizing multiple queries in distributed data stream systems. In: Proceedings of the 2nd IEEE International Workshop on Networking Meets Databases, NetDB, Atlanta, GA, USA. (2006) 25
15. Ahmad, Y., Çetintemel, U.: Network-aware query processing for stream-based applications. In: Proceedings of the 30th International Conference on Very Large Data Bases, VLDB, Toronto, Canada. (2004) 456–467
16. Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A., Vahdat, A.: Service placement in a shared wide-area platform. In: Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA. (2006) 273–288
17. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. In: Proceedings of ACM SIGMOD 2005, Baltimore, MD, USA. (2005) 13–24
18. Shah, M., Hellerstein, J., S.Chandrasekaran, Franklin, M.: Flux: An adaptive partitioning operator for continuous query systems. In: Proceedings of the 19th International Conference on Data Engineering, ICDE, Bangalore, India. (2003) 25–36
19. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based load management in federated distributed systems. In: Proceedings of Networked System Design and Implementation (NSDI), San Francisco, CA, USA. (2004) 197–210
20. Zhou, Y., Ooi, B., Tan, K.: Dynamic load management for distributed continuous query systems. In: Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan. (2005) 322–323
21. Tatbul, N., Cetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB, Berlin, Germany. (2003) 309–320
22. Tu, Y., Liu, S., Prabhakar, S., Yao, B.: Load shedding in stream databases: A control-based approach. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB, Seoul, Korea. (2006) 787–798
23. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in dynamic structured P2P systems. In: Proceedings of INFOCOM, Hong Kong. (2004)
24. Zhou, X., Nejdl, W.: Priority based load balancing in a self-interested P2P network. In: Proceedings of the 4th International Workshop on Databases, Information Systems and P2P Computing, DBISP2P, Seoul, Korea. (2006) 355–367
25. Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Computing Surveys **32** (2000) 241–299