The Stream Software Connector Design Space: Frameworks and Languages for Distributed Stream Processing

Masiar Babazadeh
University of Lugano, Switzerland
masiar.babazadeh@usi.ch

Cesare Pautasso
University of Lugano, Switzerland
c.pautasso@ieee.org

***Abstract*—In recent years we witnessed the rise of applications in which data is continuously generated and pushed towards consumers in real time through complex processing pipelines. Software connectors like remote procedure call (RPC) do not fit with the needs of such applications, for which the publish/subscribe and the stream connectors are more suitable. This paper introduces the design space of the stream software connector by analyzing recent stream processing engine frameworks and domain specific languages featuring native streaming support. On the one side, we want to classify and compare streaming systems based on a taxonomy derived from the wide range of features they offer (i.e., pipeline dynamicity and representation, load balancing and deployment flexibility). On the other side, the gaps in the design space we identify point at future research directions in the area of distributed stream processing. To do so, we gather valuable architectural knowledge in terms of architectural issues and alternatives, elicited by surveying the most important architectural decisions made by the designers of several representative streaming framework architectures.**

Keywords*—stream; software connector; architectural decisions;

I. Introduction

The notion of "stream" has become of primary importance nowadays with the emergence of applications in which data is continuously generated and pushed through complex processing pipelines towards consumers [1]. Examples range from the banking, financial sector to pervasive computing (i.e., the Internet of Things or the Web of Things [2]) as well as the increased popularity of social media. In the last decades, many attempts to natively embed the stream abstraction into software frameworks and programming languages have been proposed. One of the most recent standards is WebRTC [3], which enables streaming peer-to-peer connections between HTML5 compliant Web browsers, paving the way for the real-time Web [4].

This paper introduces a novel classification for such languages and frameworks by analyzing the most important architectural-level issues covered during their design, and by presenting which strategic design decisions have been taken to address them. These architectural decisions [5] impact the ease of use (in terms of programming abstractions) for developers as well as other important quality attributes (i.e., performance, dependability, scalability) of the resulting stream-based system. The classification can be used to compare the architecture of state-of-the-art software frameworks providing support for the stream software connector abstraction [6], which together with remote procedure calls, messaging, file transfer, and shared database provides an important alternative solution for the integration of different systems in large enterprise architectures [7]. What is specific about streaming is the ability to deliver a potentially infinite stream of messages. As we are going to discuss, depending on the design of the specific framework, streams may be used to build complex topologies and be delivered with real-time guarantees that are not commonly associated with other software connectors. As a result of this survey, we have identified several gaps in the design space of streaming software connectors that can lead to further research in next-generation streaming middleware systems.

The paper is organized as follows: Section II presents the methodology used to choose and classify the systems; III introduces the selected streaming connectors, frameworks and languages; Sections IV and V enumerate the main architectural design-time and run-time issues streaming systems should address and discusses how the architectures of the streaming systems we have selected address them. Section VI discusses the results. Section VIII concludes the paper.

II. Methodology

To gather architectural decisions on stream software connectors, we have focused on the functional characteristics of a set of representative stream processing frameworks and languages. The goal was to reconstruct and analyze the set of principal design issues that, taken together, allow us to classify and compare the capabilities and features of different kinds of streaming connectors. To organize the architectural knowledge [8], we follow a simple Issue-Based Information Systems (IBIS) meta-model [9], where we enumerate a number of design issues (or problems) and – for each issue – we present a number of architecture alternatives (or solutions) that have been chosen in the frameworks and languages we surveyed. For each alternative, we outline the benefits, challenges and concrete examples of how they are supported by one or more representative connector. We decided to divide the design issues into two categories: design-time and run-time. Design-time issues describe what has to be taken into account for a system before running it. For example, which kind of topology it will implement, how the application is described, or where is it going to be deployed. Run-time issues instead describe what should be decided for the run-time of the system, for example, if the topology is going to be flexible, if the system will tolerate faults, or balance the load. The two dimensions are orthogonal with respect to the time: deciding the type of topology, how to write it, and where to deploy happens before running the topology; concepts like dynamic adaptation, fault tolerance and load balancing are mainly run-time notions. To recover the architectural decisions, we followed a process similar to Jansen *et al.* [10] where we gathered information about each connector, testing it where possible, and harvested architectural decisions from the available documentation. In particular, we decided to include an issue in the design space only if we could find more than one alternative for it. The choice of systems to include in the survey has been based on the following criteria: 1) support for *distributed* stream processing; 2) availability of prototypes or implementations with actual real-world usage. In particular, for space reasons, we omitted redundant examples with systems that are very similar or closely related (i.e., newer systems were preferred to their predecessors). The selection process resulted in a representative but also very diverse set of stream software connectors, which allowed us to evaluate the taxonomic power of the design space. As a result, we can compare the different decisions behind the various systems and also observe how technology trends have emerged and evolved over the past 10 years.

III. Background

From an architectural perspective, the stream software connector is tightly linked to the well known pipe and filter architectural style [11]. Early architectural description languages (i.e., UniCon [12]) introduced the Pipe connector to exchange an infinite stream of elements. The stream may be composed of raw bytes (as with TCP/IP socket connections) or have a structured content (as with multimedia digital video streams). Whereas the stream would initially connect local processes or locally deployed components, very soon remote or distributed implementations of streams became available with various degrees of reliability and performance qualities.

From a data management perspective, streams have also been the fundamental abstraction of Data Stream Management Systems (DSMS) [13], which—as opposed to database management systems—feature support for consuming and generating infinite relationships by continuously running queries over one or more sources of streaming data.

As opposed to the architectural perspective given by stream software connectors, which focus on the architectural-level connection and the runtime interaction between the interfaces of software components, the data stream management system perspective is devoted more to the content flowing through the stream and dedicated to describing how to efficiently process such content, by providing reliability and scalability guarantees to deal with very large amounts of data flowing through the streaming pipeline.

We have explicitly collected representative systems from both the software architecture and the database areas, in order to attempt to provide a unified perspective on the design space presented in this paper. Given that different systems use different notations and vocabulary to describe primitives and basic building blocks of a stream processing system, in this paper, we decided to introduce a common terminology. Figure 1 shows the terms used in this paper in the context of a simple example pipeline. With the term "system" we address frameworks and/or streaming programming languages. With the term "host", we define the hardware where (part of) the streaming system is run. The term "operator" indicates a logical component of the pipeline. The term "worker" is the runtime element (i.e., process or thread) dedicated to processing the stream on the corresponding operator. For scalability/reliability, multiple workers deployed on different hosts can run the same logical operator. Workers are connected through a stream of data, whose units are the "stream elements". Operators connected by the stream form a graph according to a predefined topology.

In the rest of this section we briefly introduce the systems we selected to be part of the survey.

a) Borealis [14] is a distributed stream processing engine which inherits core stream processing functionality from Aurora [15] and distribution functionality from Medusa [16]. Aurora is a framework for monitoring applications; the high level system model is made out of operators which receive and forward data through continuous and ad hoc queries. An extension of Aurora called Aurora* [17] adds distribution and scalability to the framework. Borealis implements all the functionalities from Aurora and is designed to support dynamic revision of query results, dynamic query modification and to be flexible and highly scalable.

b) CQL [18], in contrast to Aurora and Borealis, is not a framework but a programming language and an execution engine. CQL (for *Continuous Query Language*) is a declarative language that extends SQL with the capability of querying windows over infinite streams of data and runs on the Stanford STREAM [19] runtime.

c) Discretized Streams (D-Streams) [20] is a system that provides a set of stream-transformation which treat the stream as a series of deterministic batch computations on very small time intervals. In this way, it reuses the fault tolerance mechanism for batch processing, leveraging MapReduce [21] style recovery.

d) DryadLINQ [22] enables a new programming model for distributed computing on large scale. It supports general-purpose declarative and imperative operations on datasets by the means of a high-level programming language. A DryadLINQ program is composed by LINQ [23] expressions automatically translated by the compiler into a distributed execution plan (which they call "job graph", a topology) passed to the Dryad execution platform.

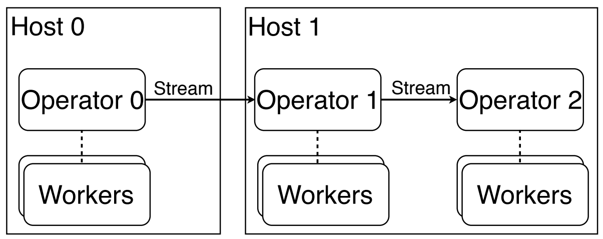

Fig. 1. Example pipeline with the terminology used in this paper.

	Issue, Alternative	StreamIt (2002)	CQL (2003)	Sawzall (2003)	Borealis (2005)	SPC (2006)	StreamFlex (2007)	DryadLINQ (2008)	S4 (2010)	Storm (2011)	WebRTC (2011)	XTream (2011)	D-Streams (2012)	TimeStream (2013)
Design Time Issues	**Topology**													
	Linear	+	+	+	+	+	+	+	+	+	+	+	+	+
	Parallel Flows	+	+	+	+	+	+	+	+	+	+	+	+	+
	DAG	+	+		+	+	+	+	+	+	+	+		+
	Arbitrary					+					+			
	Topology Representation													
	Textual, Declarative		+					+						+
	Textual, Imperative	+		+	+	+	+	+	+	+	+	+	+	+
	Deployment													
	Cluster	+		+	+	+	+	+	+	+		+	+	+
	Cloud				+	+			+	+		+		+
	Pervasive				+							+		
	Web Browser										+			
Run Time Issues	**Dynamic Adaptation**													
	Static	+	+				+						+	
	Dynamic: Operator			+	+				+	+				+
	Dynamic: Topology					+		+			+	+		+
	Fault Tolerance													
	Replication				+	+								
	Reconfiguration							+	+	+			+	+
	Load Balancing													
	Load Shedding		+		+		+							
	Dynamic Reinitialization	+												
	Dynamic Adaptive Controller							+	+	+				+

TABLE I. SUMMARY OF THE DESIGN DECISIONS (+) OVER THE STREAM CONNECTOR DESIGN SPACE

e) The Simple Scalable Streaming System (S4) [24] is a general-purpose, distributed platform similar to Storm, developed by Yahoo! and inspired by the Actor model [25] and MapReduce. It allows programmers to develop applications that process unbounded streams of data. Programmers can build topologies out of Processing Nodes (PNs) that host Processing Elements (PEs, operators). Each PE is associated with a function and with the type of event that it consumes.

f) Sawzall [26] is a procedural domain-specific programming language built upon MapReduce. It was introduced by Google to process large batches of log records. Each Sawzall script takes as input a single log record to be processed by operators deployed on multiple machines; the output is emitted in tabular form.

g) Storm [27] is a distributed real-time computational environment, free and open source, originally developed by Twitter. Information sources and manipulations are defined by custom created "spouts" (operators that produce a stream) and "bolts" (operators that receive streaming elements), used to allow distributed processing of streaming data. These are very similar to MapReduce jobs, but they can theoretically run forever. Every spout and bolt can be run by one or more worker, in which the core of a Storm computation is executed.

h) StreamFlex [28] is a programming model for high-throughput stream processing in Java. It extends the Java Virtual Machine with transactional channels and type-safe allocation while providing a stricter typing discipline on the stream components of the code.

i) StreamIt [29] is a programming language and a compilation infrastructure. A StreamIt program is a hierarchical composition of constructs called Filter, Pipeline, SplitJoin or FeedbackLoop which restrict the topology and the message rate.

j) Stream Processing Core (SPC) [30] is a middleware for distributed stream processing targeting data mining applications. It supports applications that extract information from multiple digital data streams. Applications are composed by Processing Elements (operators) which implement application-defined operators connected by stream subscriptions.

k) TimeStream [31] is a distributed system designed for continuous processing of big streaming data on large cluster of machines. It supports failure recovery and dynamic reconfiguration to face load changes. TimeStream tries to combine MapReduce-style and streaming database systems into one framework. It is designed to preserve the programming model of StreamInsight [32] (a platform used to develop and deploy complex event processing applications) and thus can be used to scale any StreamInsight application to large clusters without modification.

l) Web Real-Time Communication (WebRTC) [3] is an API drafted by the World Wide Web Consortium (W3C). It enables browser-to-browser connectivity for applications such as voice calls, video chat, and peer-to-peer file sharing. It makes is possible to use the stream connector directly between Web browsers.

m) XTream [33], [34] is a platform to support the design of data processing and dissemination engines at Internet scale. XTream defines a model for building applications with personal information streams. It is composed by *slets*, which are what we call operators, and can appear in three different aspects: α-slets (inputs), ω-slets (outputs) and π-slets (middle of the topology). Slets can be loaded and unloaded at runtime, giving the system high flexibility. One of the most interesting aspects of XTream is the ability to instantiate and run other kind of topologies (i.e. a CQL topology) within one of its *slets*.

Table I summarizes the stream connector design space as well as the design decision taken by the designers of the analyzed systems. The issues and alternatives are introduced in the following two Sections, while Section VI discusses Table I. Given the design decisions, we would like to (1) observe the evolution over time of the decisions taken for each issue; (2) discuss the decisions considered most and least viable while presenting their impact on the system; and (3) predict possible future research directions while analyzing the trending design decisions.

IV. Design-time Issues

This section describes the architectural issues that affect how the stream connectors are used to design a stream-based application. We will first look at the topology issue which describes the arrangement rules of the operators in a pipeline. Next, we show how topologies may be represented, that is, how to build operators and wire them together. Finally, we look at where these topologies may be deployed.

A. Design Issue: Topology

The topology of a streaming system describes how the stream flows through multiple operators and in which order these operators process one, or possibly multiple streams. As a design issue, this has strategic importance as it impacts the overall expressiveness of the streaming framework/language. The most frequent alternatives we observed are: linear, parallel flows, direct acyclic graph (DAG), or arbitrary topology.

1) Linear: The linear pipeline is the simplest kind of topology. There are no splits or joins along the pipeline. When an operator receives a stream element, it processes it and forwards the result only to the single operator following it downstream.

Benefits: The simplicity of a linear pipeline is sufficient for applications that can contain all branching decisions within the operators along the pipeline and no alternative flows are needed to process different types of stream elements based on intermediate results.

Challenges: This kind of pipeline has an underlying problem of limited expressiveness. Developers may set up long linear topologies but are not allowed to create branches in the dataflow, thus this approach fits only a limited subset of applications.

Example: Every connector we analyzed is able to construct this kind of topology.

2) Parallel Flows: An extension to the previous alternative can be built by following a data parrallelism approach: a single data source where the stream starts, many identical linear topologies that flow out of it and one final operator where data is aggregated from the parallel stream flows.

Benefits: Having parallel flows with an initial operator that scatters the work and possibly a final operator that gathers it can help to cope with workload variations or high data rates, for example exploiting a multicore architecture [35].

Challenges: As before, this approach lacks of expressiveness as the data flows along identical parallel linear topologies. Depending on the application semantics, it may be important to ensure that the order of stream elements is preserved once the parallel flows are merged.

Example: This kind of pipeline is usually found when MapReduce-style data parallelization is combined with jobs that are themselves parallelized along a linear pipeline.

3) Directed Acyclic Graph (DAG): In this case each operator may be connected to one or more operators. The data exchange may happen because of some branching condition, or following a particular routing algorithm. The only constraint of this alternative is not having cycles in the topology.

Benefits: With a DAG topology, users can structure more complicated flows. For example, work may be split across different operators and then the result may be gathered back (e.g., with operators starting a MapReduce-like computation), which a linear pipeline would not allow. Branches can be task based, or heuristic (i.e. depending on the result of a computation perfomed at a previous operator in the topology).

Challenges: Giving the possibility to create acyclic topologies implies introducing complexity in the system to be built. Users must be given the ability to decide how to split the work and the possibility to "reduce" it eventually. Heuristic choices must be admitted as well.

Example: All the analyzed systems can run a DAG topology, except D-Stream and Sawzall.

4) Arbitrary: A topology with an arbitrary topology gives enough expressive power to setup any kind of topology for the stream flowing through a directed graph of operators and workers. As opposed to the DAG topology, the possibility to create cycles needs to be handled carefully. With cycles, data may flow indefinitely many times through the same operator(s) before reaching the end of the topology.

Benefits: This alternative provides the highest level of expressiveness, whereby complex topologies can be built that feed the output of downstream operators back into upstream operators. These cycles in the topology may be useful to process sensor data streams in control loops, or to perform recursive computations.

Challenges: Cycles may cause deadlocks of operators waiting to receive their own results as input. Alternatively, stream elements may flow indefinitely throughout the topology. A system supporting feedback loops need to address both of these issues.

Example: The only connector supporting this alternative is WebRTC.

B. Design Issue: Topology Representation

To use the stream connector, it is necessary to define and link together two or more operators into a topology. To do so, different alternative representations have been proposed, with varying level of abstraction and expressiveness. We distinguish between two main categories of textual representations: declarative languages (i.e., rule-based) and imperative languages. A visual representation can be added on top of both the representations, increasing the level of abstraction.

1) Textual Representation, Declarative: A declarative representation lets developers organize topologies by building the structure of the system without describing its data flow. Side effects are minimized or eliminated by characterizing what the program should carry out as a result and not describing how to obtain that result through a combination of basic processing steps.

Benefits: Focusing on the specification of the result helps to abstract the underlying implementation. The DSMS can use the declarative specification to constrain or to guide the runtime optimization of the topology. Existing declarative languages can be extended with the notion of stream (e.g., continuous queries, windows and stream-to-relation mappings).

Challenges: Given the high abstraction level, the visibility into the actual topology being executed may suffer. Additionally, stateful operators are challenging to apply over an infinite data stream. To do so, these operators (e.g., aggregation) are usually applied over windows or limited-size buffers.

Example: This alternative is implemented by CQL, DryadLINQ and TimeStream. DryadLINQ uses LINQ (which also supports declarative statements) while CQL works on top of STREAM [36], a Data Stream Management System (DSMS) and has borrowed much of its representation from the Structured Programming Language (SQL), which is a special-purpose declarative programming language. The following example shows a CQL query. It returns the total cost of orders fulfilled over the last day by clerk "Sue" for customer "Joe".

```
Select Sum(O.cost)
From Orders O, Fulfillments F [Range 1 Day]
Where O.orderID = F.orderID And F.clerk = "Sue"
    And O.customer = "Joe"
```

In this case there are two different streams. The first is `Orders (orderID, customer, cost)` which is a stream of orders with an order ID, the customer that made the order and the total cost, while the second is `Fulfillments (orderID, clerk)`, a stream of fullfilled orders that includes the fulfilled order ID and the clerk that fulfilled the order. The query is performed on the "1 Day" window and the result is the total income obtained by "Sue" in the last day on the orders submitted by "Joe". TimeStream adopts the declarative programming model used by the Microsoft StreamInsight framework [37], which is a LINQ-dialect.

2) Textual Representation, Imperative: Stream processing is also possible with traditional imperative programming languages, which can make use of specific streaming libraries or include specific streaming constructs.

Benefits: The same language is used to program the operators as well as to configure the topology: existing code can be repurposed as stream operator.

Challenges: A textual specification of the graph building the topology may not be as easy to grasp as its corresponding visual representation. The composition and the component boundaries become blurred.

Example: Systems implementing this alternative are D-Streams, DryadLINQ, S4, Storm, StreamFlex, StreamIt, TimeStream, WebRTC and XTream. D-Streams makes use of Scala; with StreamFlex, Storm, S4 and XTream users can code directly in Java the operator functionality, while TimeStream, despite adopting the StreamInsight programming model, supports user-defined functions, aggregators and operators written in an imperative language. With WebRTC one can setup a data stream between two Web browsers using JavaScript. The following example shows the minimal StreamIt program introduced in the StreamIt Cookbook.

```
void->void pipeline Minimal {
    add IntSource;
    add IntPrinter;
}
void->int filter IntSource {
    int x;
    init {
        x = 0;
    }
    work push 1 {
        push(x++);
    }
}
int->void filter IntPrinter {
    work pop 1 {
        print(pop());
    }
}
```

The example instantiates a filter that produces a series of numbers and forwards it to the second filter which consumes them. StreamIt embeds directly the topology inside the code with the `pipeline` and `add` constructs and defines the operator functionality within a `filter`. The `push` and `pop` functions are used respectively to push the result of the operator in the output queue and pop received messages from the input queue.

C. Design Issue: Deployment Environment

The stream connector allows components deployed on different environments to communicate while hiding the distributed nature of the interaction. The deployment issue distinguishes the type of environments that are specifically targeted by a streaming framework. The following Section discusses the four most common non-local types: cluster, cloud, pervasive, and browser-based.

1) Cluster: A physical cluster of machines is the standard target deployment for a streaming system that requires the computing and storage capacity of more than one machine.

Benefits: Analyzing a high-throughput stream of data is a task that may require a significant amount of computing power. A cluster is a cost-effective solution to scaling out the available computing resources, with the advantage that an optimized local network interconnect can be used to reduce the latency and increase the bandwidth available.

Challenges: A cluster of machines has to adopt good load sharing and load balancing strategies to work. Unlike a single machine, where parallelization is bound by the processor cores, a cluster can parallelize the work by one or two orders of magnitude, depending on the size of the cluster. Thus machines have to work cooperatively to achieve the best parallelization of the streaming topology possible, that is, depending on the kind of data and the kind of computation, work along the topology has to be balanced across multiple operators. It is challenging to do so with arbitrary stream operators and with varying data rate in the input streams.

Example: Every system mentioned in this survey is able to run on a cluster of dedicated machines except CQL and WebRTC. In principle, a system should be able to treat each machine as an operator where it can instantiate workers and perform a job.

2) Cloud: Conceptually, this alternative is very similar to the previous one, with the key difference that the stream is running across a distributed cluster of virtual machines, deployed across one or more data centers.

Benefits: Adding resources is not a matter of physically wiring machines into a cluster rack, but can be done in seconds by renting and initializing a virtual machine. Every reachable server on the cloud is now a potential operator for the computation, which can thus reach an even larger scale. Parts of the topology can be deployed across different regions, bringing the edges of the stream closer to the producers/consumers, which may be located outside of the cloud data center. The streaming system can potentially take advantage of the Cloud virtualization layer to deal with load balancing and reliability issues.

Challenges: Since there is limited control over the placement of virtual machines in the cloud data centers, less guarantees are provided over the actual network conditions. This implies not only increased latencies, but also may make it more challenging to ensure the Quality of Service of the resulting stream.

Example: This alternative is currently supported by Borealis, S4, SPC, Storm, TimeStream and XTream. The idea is similar to the cluster of machines, with the only difference that the machines now are in the cloud. Systems implemented with this alternative have workers with socket connections able to connect and send work to remote virtual machines.

3) Pervasive: The pervasive approach takes the distributed deployment to the extreme with the possibility of running parts of the topology directly on the stream data sources, such as sensors or embedded/mobile devices.

Benefits: Direct access to streaming data sources allows the streaming framework to optimize the flow of incoming data (i.e., only interesting data samples are inserted into the topology at the rate the topology can handle) as well as avoid inefficient polling (where the data source must be periodically sampled for new information).

Challenges: Although hardware is constantly improving as the time passes by, small hardware and sensors are not yet suited for heavy computation. Thus, parts of the topology may still need to be deployed using the cloud, or on a dedicated cluster of computers.

Example: Aurora, the predecessor of Borealis, was developed to access and stream data from sensors. XTream, with the concept of α-slets and ω-slets can incorporate pervasive devices.

4) Web Browser: With the advent of WebRTC, browser to browser streaming of data has become possible through a standard protocol. This way, browsers may become part of a topology, playing the role of data producer, intermediate data processor or data consumer.

Benefits: Establishing a direct browser-to-browser communication stream opens many opportunity for new kinds of Web applications (i.e., videochat systems), especially considering that HTML5 compatible browsers also run on the latest generation of mobile devices.

Challenges: A browser cannot easily discover the network address of other browsers without support of a dedicated name and directory service.

Example: Using WebRTC, to connect to another browser, a browser has to generate an ID and request a Channel token from a Web server, which in turn requests a channel and a token for the client's ID from the Channel API. The token is sent back to the client, which opens a socket and starts listening. A second client is going to follow the same procedure so that it can be redirected from the server on the same channel, where the direct stream connection is estabilished.

V. Run-time Issues

Depending on the targeted deployment environment, a stream connector will impact the run-time behavior of a software architecture in terms of its flexibility and reliability. In this section, we first look at the flexibility of operators and topologies, that is, if they can be dynamically reconfigured, migrated or replicated to provide additional processing capacity and fault tolerance. We then discuss the impact on the fault tolerance and load balancing capabilities of the different connector variants.

A. Design Issue: Dynamic Adaptation

Flexibility determines if the topology has the ability to be reconfigured at runtime. A static topology must be stopped before it can be reconfigured. For a dynamic topology we distinguish two orthogonal alternatives: whether the topology can be changed or whether the configuration changes affect the operators (e.g., by increasing or decreasing the resources allocated to execute them).

1) Static: A static topology does not change at runtime. Once the topology and deployment configuration are given and the stream of data starts flowing, there is no way to change the topology by adding or removing operators from it.

Benefits: The runtime system is simplified, since there is no need to introduce support for adding or removing operators.

Challenges: Dynamic behavior (e.g., reordering or removal of operators) can be achieved by appropriately using branching operators, whereby parts of the pipeline can be skipped. This however requires additional effort by the stream developer, since all changes need to be planned for in advance.

Example: CQL (STREAM), D-Streams, StreamFlex and StreamIt cannot modify the topology configuration after the stream starts flowing through it.

2) Dynamic, Operator: This alternative allows to adapt the operator configuration at runtime. The configuration may include the parallelism level within the operator (i.e., the number of workers dedicated to run it), the location and the set of hosts allocated to run the workers, and the corresponding routing/load balancing configuration.

Benefits: In presence of failures, the topology has the possibility to repair the error by, for example, routing the traffic on a different operator or restarting, recovering or migrating the failed operator, while keeping the logical topology between operators unchanged.

Challenges: Operator migration needs to be accomplished without data loss. Some form of dynamic binding is required because the route followed by the stream is not fixed, thus increasing the complexity (and possibly the overhead) of the stream connector.

Example: Among the systems analyzed, Borealis, S4, Sawzall, Storm and TimeStream use this kind of dynamicity to handle failures, as discussed more in detail in Section V-B.

3) Dynamic, Topology: With a fully dynamic topology, operators may be re-arranged, added and removed from the topology at runtime, to form new topologies without stopping the stream.

Benefits: This high degree of dynamicity helps to introduce a form of live programming [38] with stream processing, where in case the topology needs to be extended or adapted to changes into its input, it is possible to do so without stopping the whole stream.

Challenges: The atomicity and the semantics of all change operators need to be defined. It is possible that by removing an intermediate operator, the data cannot be immediately routed to the following one, thus making the whole topology temporarily inconsistent and in need of further repair actions.

Example: DryadLINQ, SPC, WebRTC, TimeStream and XTream are the systems implementing this alternative. WebRTC does not constrain how and when the browsers should be connected. Thus, any kind of operator reordering, addition and removal can be executed. If not handled, messages may be lost in the process. SPC and XTream on the other hand support connecting new operators to the topology at run time, thus the topology can be extended while the stream flows. DryadLINQ, through the Job Manager component, is able to modify the topology according to user-supplied policies. TimeStream supports dynamic reconfiguration of the topology with a restriction that such reconfiguration must be a substitution of two equivalent sub-DAGs.

B. Design Issue: Fault Tolerance

Fault tolerance is a very important aspect for a stream processing system. If an operator fails, it can compromise the whole topology, thus failures have to be handled quickly. While most frameworks support some form of fault tolerance, CQL and StreamIt do not address the problem. Sawzall relies on the underlying MapReduce infrastructure. Borealis takes advantage of replication, while S4, Storm and TimeStream use ad-hoc reconfigurations. WebRTC gives the choice between a reliable data channel (TCP) or not (UDP). XTream relies on the intrinsic dynamicity of its topology to cope with faults.

1) Replication: The concept of replication involves information sharing to ensure the consistency between redundant resources to improve accessibility, reliability and fault tolerance [39]. In streaming systems this concerns replicating workers in a way that the stream can bypass the failure of any of them.

Benefits: The main benefit of employing replication is the possibility to deviate the flow of the stream from the failed operator to one of its replicas. Replication is a good approach for lightweight topologies, where the added computational and communication cost of replicating operators remains limited.

Challenges: In addition to the extra resource consumption, replication suffers from additional problems [40] that involve mainly maintaining consistency of replicated state. If an operator fails, the recovery protocol should not invalidate the consistency of the replicas.

Example: In Borealis, when a failure of a worker is detected, the system tries to find an alternative upstream replica to continue processing the stream. In order to do so, however, the upstream replicas must be consistent with each other. Borealis defines an operator called *SUnion* that takes as input multiple streams and outputs a stream with tuples ordered in a deterministic way so that all the replicas process exactly the same input. Support for replication is also mentioned as future work for SPC.

2) Reconfiguration: Reconfiguration is the process of updating the topology configuration as a consequence of a failure. When a failed operator is detected, it is immediately substituted or re-launched by the underlying runtime.

Benefits: There is no need to keep an active replica synchronized, since failed operators are restarted or substituted automatically.

Challenges: The automatic process of reconfiguration of the topology needs a reliable monitoring component which constantly checks the topology for failures.

Example: In Storm, failed workers are automaticaly restarted by the topology supervisor. If the operator keeps failing and is incapable of beginning to consume its dispatched stream elements, the *Nimbus* daemon will reassign it to another host. In D-Stream, lost stream elements are recomputed while an operator is being reinitialized to speed up the recovery of the topology. Also DryadLINQ supports the automated recomputation of results produced by a failed worker, since these are re-executed by the Job Manager component. TimeStream uses the concept of Resilient Substitution to replace failed operators, possibly by restarting them on a different machine. S4 can reconfigure a topology splitting the operators deployed on a failed host among the remaining available execution resources.

C. Design Issue: Load Balancing

A balanced topology can process the stream at a regular data rate (or throughput) by making good use of the available hardware resources to run the operators. In some applications, it is difficult to predict the computational cost of each operator (which may depend on the values of each stream element) and also the rate of the incoming stream elements may change dynamically and unpredictably. To deal with these unbalances in the flow, different frameworks propose different approaches, going from load shedding for static systems to more adaptive or dynamic alternatives.

1) Load Shedding: Assuming the application can tolerate data loss, load shedding will reduce the stream data rate by dropping some stream elements along the topology. This change of semantics results in a best-effort streaming system, where no guarantees are made on the data reaching the end of the topology. Since saturation is reached without overloading the system, it is possible to give guarantees on the overall throughput.

Benefits: Load Shedding is the simplest approach to deal with congestion in case the application can tolerate loss of stream elements which are dropped randomly along the topology where bottlenecks occur.

Challenges: If we can assume that some stream elements are more important than others, then load shedding can be refined with the ability to control which elements are dropped to avoid doing so at random. This requires the connector to know more information about the application semantics of the stream.

Example: Aurora and Borealis implement two different load shedding methods based on Quality of Service guarantees. The first one consists into dropping tuples from queues buffering stream elements in front of operators whose output can tolerate their loss. By dropping randomly selected tuples at strategic points in the topology, this approach effectively reduces the workload. The second approach is called Semantic Load Shedding and consists of dropping the least important tuples. Importance is determined by the utility interval, which is computed by observing the QoS of the application.

2) Dynamic Reinitialization: The whole topology is reinitialized by sending a special token through it that informs all operators that they have to flush the remaining stream elements and then re-initialize and re-allocate enough resources.

Benefits: Data loss is avoided since there is a clear mechanism for dynamically changing the resources allocated, which however requires to temporarily flush the topology.

Challenges: The stream protocol is more complex, since it should both carry the normal data flow as well as special control messages to re-initialize the topology. The decision to re-initialize can be determined automatically by a controller or it requires manual intervention.

Example: When a topology needs to be modified, the protocol uses a `init` message. When such message is received by a worker, it executes once more the initialization code of its operator and can adjust the amount of resources allocated to run it. This process is only theoretical, as it has been described but never implemented. We decided nevertheless to keep it as a feature offered by the system.

3) Dynamic Adaptive Control: A controller is introduced to balance the load along the topology. If there are bottlenecks, the controller takes action by, for example, adding more hosts or workers to the topology, or trigger some optimizations at the worker level.

Benefits: Through automated load balancing, the controller can automatically trade off resource consumption against the performance of the stream.

Challenges: Real-time monitoring of the system is required, and the controller may need to be tuned to obtain good performance. The dynamic reconfiguration of the topology should be safe (to avoid data loss).

Example: There have been many different approaches to support dynamic adaptive control. For example, Storm allows to modify the number of operators at runtime by means of a controller or a GUI/command line administration tool. TimeStream uses its resilent substitution feature to deal with bottleneck issues by automatically changing the number of hosts on which a given operator of the topology is deployed. Likewise, if an operator becomes the bottleneck, its performance can be improved by boosting its parallelism. DryadLINQ exploits hooks in the Dryad API to mutate on runtime the topology to improve the performance by aggregating operators and reducing I/O operations.

VI. Discussion

A. Relationships

Figure 2 shows some of the most important relationships between the issues and alternatives of the design space. The issue with more impact on other issues and alternatives is the dynamic adaptation. Dynamicity at the operator or topology levels helps to achieve fault tolerance in general (as also shown by Table I) while for what concerns load shedding, dropping stream elements is the only alternative available for a static topology. Instead, flexibility at the operator-level avoids loosing stream elements and can allow to balance the load with reinitialization, or an adaptive controller. Moreover, a very flexible topology can support volatile operators, like Web browsers. On the other hand, running a stream across multiple Web browsers can only be supported by a very flexible topology.

B. Evolution

We decided to divide the analyzed systems in three different generations based on their age and features. First generation systems (StreamIt, CQL, Sawzall) share the following commonalities. Topologies are parallel linear or DAG, while

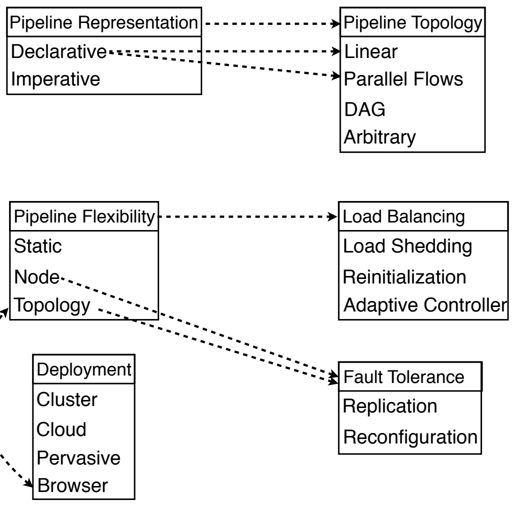

Fig. 2. Relationships among issues and alternatives.

the representation is purely with code (declarative for CQL, imperative for StreamIt and Sawzall). The only target deployment is the cluster of machines (with the exception of CQL). Given their age, it's not surprising that no cloud deployment option has been foreseen. These systems do not present a flexible topology and do not address fault tolerance. As for load balacing, CQL uses load shedding, while Sawzall bases its own on MapReduce.

Borealis defines itself a "second-generation distributed stream processing engine" [14]. We also include in this category SPC, StreamFlex, DryadLINQ and S4. Topologies are more complex, the concept of deploying on the cloud has been introduced as well as the idea of deploying on sensors (pervasive deployment). Topologies also become more flexible at the operator level (Borealis, S4, Sawzall) as well as at the structure level (SPC, DryadLINQ). Fault tolerance is introduced in these systems with replication and reconfiguration. Balancing the load sees a shift from the load shedding approach towards a dynamic controller.

For the latest state-of-the-art streaming frameworks, we presented a heterogeneous sample by including Storm, WebRTC, XTream, D-Streams and TimeStream. The topology again includes DAGs, with the exception of D-Streams and WebRTC that also supports an arbitrary topology. The representation has narrowed to imperative, while the distribution is more in favor of cluster of machines and the cloud (with the exception of XTream and D-Streams). WebRTC, again, is a special case which enables to deploy operators on a Web browser. The system is dynamically adaptable at the operator level (Storm, TimeStream) or at topology level (WebRTC, TimeStream, XTream). D-Streams is again the only exception because of its MapReduce nature. Fault tolerance is tackled with reconfiguration, and load balacing presents the use of a controller as standard solution, definitely abandoning load shedding. We can define three different groups: Storm, XTream and TimeStream leverage the work proposed by StreamIt. D-Streams is an evolution of MapReduce, while WebRTC is a new technology to exploit streams on a web browser.

The big picture shows an initial trend where the target hardware architectures were fixed (cluster of machines), offering no flexibility and barely any topology complexity. Fault tolerance was mostly not supported, while load balacing was achieved by relying on the underlying runtime platform. The trend shifts over time, by first adding more deployment options and fault tolerance and finally shifting towards (almost) arbitrary and dynamic topologies, dynamic reconfiguration both for load balancing issues and fault tolerance and almost the same deployment options. It is interesting to observe that this trend towards more and more runtime flexibility is obtained while the representation alternative is constrained towards the use of imperative languages.

C. Outlook

Given this overview of the evolution of the design space, we can speculate on future trends. A traditional system would tend to prefer a specific set of design alternatives to fit the needs, while a general-purpose system would give more degrees of freedom. Support for DAG topologies appears to be an adequate level of expressiveness for most applications. With the rise of hardware like Arduino, RasperryPi, Beaglebone or Tessel, we expect the pervasive deployment alternative to be re-explored while the pervasiveness of handheld and tablets capable of running an HTML5 Web browser suggests that the WebRTC APIs might be enable new kinds of stream-based applications on the Web. Fault tolerance is likely to become even more critical with the latest reconfiguration trend that makes use of a centralized controller, which will also balance the load (by adding or removing workers). Thus, we expect to see systems where dynamicity will be featured both at the operator and at the topology level. We also expect future systems to be much more decentralized. In the past we had DSMS deployed on a single-machine, which evolved into systems distributed across a cluster of machines. These turned into stream-based architectures running on virtualized clusters on the Cloud, as well as spread across pervasive deployments and more recently Web browsers. We foresee future systems not only to be more and more distributed, but also starting to introduce decentralized control architectures (i.e. more than a single controller for fault tolerance and load balancing), to better support very big topologies and to swiftly deal with rapid changes in the environment.

VII. Related Work

This paper takes inspiration from [6], which describes a taxonomy for software connectors. Our work is entirely focused on the stream software connector and gives a different classification of the design issues and alternatives of a large variety of streaming frameworks and languages. Many research issues specific to Data Stream Management Systems have been collected in this 2003 survey [13]. A more recent analysis specific to DSMS can be found in [41], covering the functional and processing model, deployment model, interaction model (push vs. pull) and date and time rules. Optimizations for stream processing have been also analyzed in this 2011 survey [42], which consolidates the prior optimizations work and provides a guide for users and implementors. Our work is complementary, both in terms of the different set of design issues we discuss as well as the broader set of systems included which is not limited to DSMS. A historical perspective on the evolution of stream processing systems can be found in [43]. Data Streams: Models and Algorithms [44] is a complementary study, focused on stream mining algorithm study, classification and analysis of data with a chapted dedicated to load shedding. A more network-oriented perspective can be found in this work [45] where the authors analyze subset of peer-to-peer streaming systems (Octoshape, SopCast, TVAnts and TVU networks) and test if they are suitable for mobile usage. A similar decision-centric methodology to describe the design space of domain-specific languages was taken in [46], [47].

VIII. Conclusion

This paper collects 6 design issues and 18 reusable design alternatives that cover a significant portion of the design space for the stream software connector. To reconstruct this design space, we analyzed 13 frameworks and programming languages that have been introduced over the last 10 years. The result is a classification and a description of the evolution over time of very different types of technologies to support the abstraction of the stream software connector. From our analysis, we made some predictions on likely future directions for the evolution of this important software connector: we expect to see more distribution for the deployment of the operators, with more decentralization in the control structure of the topology.

Acknowledgment: The work is supported by the Hasler Foundation with the Liquid Software Architecture (LiSA) project. We are grateful for the invaluable feedback of Gustavo Alonso, Zhengping Qian, Robert Soulé, Bill Thies and the anonymous reviewers.

References

[1] E. Della Valle *et al.*, "It's a Streaming World! Reasoning upon Rapidly Changing Information," *IEEE Intelligent Systems*, vol. 24, no. 6, pp. 83–89, 2009.

[2] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource-oriented architecture and best practices." in *Architecting the Internet of Things*, 2011, pp. 97–129.

[3] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "Webrtc 1.0: Real-time communication between browsers," *Working draft, W3C*, 2012.

[4] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: building a CDN from client web browsers," in *Proc. of EuroSys*, 2013, pp. 281–294.

[5] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proc. of WICSA*, 2005, pp. 109–120.

[6] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proc. of ICSE*, 2000, pp. 178–187.

[7] M. Fowler and R. Parsons, *Domain-specific languages*. Addison-Wesley, 2010.

[8] P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," in *Proceedings of the Second international conference on Quality of Software Architectures*, 2006, pp. 43–58.

[9] W. Kunz and H. W. Rittel, *Issues as elements of information systems*. University of California Berkeley, California, 1970, vol. 131.

[10] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, pp. 536–557, 2008.

[11] M. Shaw and D. Garland, *Software Architecture: Perspectives on an Emerging Discipline*, 1996.

[12] M. Shaw *et al.*, "Abstractions for software architecture and tools to support them," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 314–335.

[13] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[14] D. J. Abadi *et al.*, "The Design of the Borealis Stream Processing Engine," in *Proc. of CIDR*, 2005, pp. 277–289.

[15] D. Carney *et al.*, "Monitoring streams: a new class of data management applications," in *Proc. of VLDB*, 2002, pp. 215–226.

[16] S. Zdonik *et al.*, "The aurora and medusa projects," *IEEE Data Engineering Bulletin*, vol. 26, no. 1, pp. 3–10, 2003.

[17] M. Cherniack *et al.*, "Scalable Distributed Stream Processing," in *Proc. of CIDR*, 2003.

[18] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[19] R. Motwani *et al.*, "Query processing, approximation, and resource management in a data stream management system," in *CIDR*, 2003.

[20] M. Zaharia *et al.*, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proc. of the 4th USENIX conference on Hot Topics in Cloud Computing*, 2012, pp. 10–10.

[21] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[22] Y. Yu *et al.*, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. of OSDI*, 2008, pp. 1–14.

[23] E. Meijer, "The world according to linq," *Queue*, vol. 9, no. 8, pp. 60:60–60:72, 2011.

[24] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. of the 2010 IEEE International Conference on Data Mining Workshops*, 2010, pp. 170–177.

[25] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[26] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming Journal*, vol. 13, no. 4, pp. 277–298, 2005.

[27] (2011) Storm, distributed and fault-tolerant realtime computation. [Online]. Available: http://storm-project.net/

[28] J. H. Spring *et al.*, "Streamflex: High-throughput stream programming in java," in *Proc. of OOPSLA '07*, pp. 211–228.

[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proc. of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[30] L. Amini *et al.*, "Spc: a distributed, scalable platform for data mining," in *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, 2006, pp. 27–37.

[31] Z. Qian *et al.*, "Timestream: reliable stream computation in the cloud," in *Proc. of EuroSys*, 2013, pp. 1–14.

[32] M. H. Ali *et al.*, "Microsoft cep server and online behavioral targeting," *Proc. VLDB*, vol. 2, no. 2, pp. 1558–1561, August 2009.

[33] M. Duller and G. Alonso, "A lightweight and extensible platform for processing personal information at global scale." *J. Internet Services and Applications*, vol. 1, no. 3, pp. 165–181, 2011.

[34] M. Duller, J. S. Rellermeyer, G. Alonso, and N. Tatbul, "Virtualizing stream processing," in *Proc. of Middleware*, 2011, pp. 269–288.

[35] M. I. Gordon *et al.*, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. of ASPLOS*, 2006, pp. 151–162.

[36] A. Arasu *et al.*, "Stream: the stanford stream data manager (demonstration description)," in *Proc. of SIGMOD*, 2003, pp. 665–665.

[37] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer, "The extensibility framework in Microsoft StreamInsight," in *Proc. of ICDE*, 2011, pp. 1242–1253.

[38] B. Burg *et al.*, "1st international workshop on live programming (live 2013)," in *Proceedings of ICSE '13*, pp. 1529–1530.

[39] O. Wolfson *et al.*, "An adaptive data replication algorithm," *ACM Trans. Database Syst.*, vol. 22, no. 2, pp. 255–314, Jun. 1997.

[40] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of SIGMOD*, 1996, pp. 173–182.

[41] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[42] M. Hirzel *et al.*, "A catalog of stream processing optimizations," Tech. Rep., 2011.

[43] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[44] C. C. Aggarwal, Ed., *Data Streams - Models and Algorithms*. Springer, 2007, vol. 31.

[45] J. Peltotalo *et al.*, "Peer-to-peer streaming technology survey," in *Proc. of the 7th International Conference on Networking*, 2008, pp. 342–350.

[46] U. Zdun and M. Strembeck, "Reusable architectural decisions for DSL design: Foundational decisions in DSL projects," in *Proc. of EuroPLoP*, 2009.

[47] S. Aghaee, M. Nowak, and C. Pautasso, "Reusable decision space for mashup tool design," in *Proc. of EICS '12*, pp. 211–220.