# Optimal Component Composition for Scalable Stream Processing

Xiaohui Gu, Philip S. Yu
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{xiaohui, psyu}@ us.ibm.com

Klara Nahrstedt
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801
{klara}@ cs.uiuc.edu

## Abstract

*Stream processing has become increasingly important with emergence of stream applications such as audio/video surveillance, stock price tracing, and sensor data analysis. A challenging problem is to provide optimal component composition in a distributed stream processing environment. The goal of optimal component composition is to achieve load balancing subject to multiple function, resource, and quality-of-service (QoS) constraints while composing stream applications. In this paper, we present an adaptive composition probing (ACP) approach to the problem. Different from previous work, ACP provides a new hybrid approach that combines distributed composition probing with coarse-grain global state management. Guided by the coarse-grain global state information, ACP selectively probes a subset of candidate components to discover an approximately optimal component composition. Further, ACP is self-tuning, which can adaptively adjust the number of probes to maintain a specified composition performance target (i.e., composition success rate) in a dynamic stream environment. While the optimal component composition problem is NP-hard, our ACP approach provides an adaptive polynomial approximation solution. We have conducted extensive simulation experiments to show the efficiency, scalability, and adaptability of the ACP approach by comparing with other alternative solutions.*

## 1 Introduction

Emerging applications, such as trade surveillance for security fraud, network traffic monitoring for intrusion detection, sensor data analysis, and audio/video surveillance, call for sophisticated real-time processing on data streams. In these applications, data streams are continuously pushed to a stream processing system, where they are processed by self-contained stream processing elements called *components*. Each component provides an atomic stream processing function such as filtering, aggregation, and correlation. Because stream applications are inherently distributed, stream processing should operate in a distributed fashion. Moreover, distributed stream processing systems provide better scalability and availability for resource-intensive and quality-sensitive stream processing applications. Generally, a distributed stream processing system consists of a collection of networked servers, each of which can provide a number of stream processing components. An interesting problem in such a distributed stream processing system is to dynamically compose a quality-aware and resource-efficient stream processing application from the system's currently available components. However, it is challenging to find the optimal solution for the problem since (1) the problem is NP-hard [3], and (2) up-to-date resource states of distributed hosts (e.g., CPU, memory) and quality-of-service values of different components (e.g., response time, loss rate) are required.

Stream processing has drawn much research attention recently (e.g.,[12, 1, 11, 2]). Previous work has addressed various problems such as load shedding [12], load migration [1], and operator partition [11] in a stream processing environment. However, little research has been done to address the optimal component composition problem that is especially important for on-demand stream processing service provisioning. Although component composition has been studied under different context (e.g., [9, 13, 10, 7, 8]), previous research falls short in addressing the optimization issues in component composition, which is especially important for real-time stream processing systems.

In this paper, we present a novel *adaptive composition probing* (ACP) approach to provide a self-tuning approximation solution for the optimal component composition problem. The ACP scheme first discovers a number of *good* candidate component compositions using a limited number of state collection probes. Then, ACP selects the best component composition that achieves best load balancing from the discovered good candidate compositions. ACP addresses two key decision-making problems in composition probing: (1) how many candidate components to probe, and

(2) which candidate components to probe. For the former problem, ACP adaptively decides the number of probed candidate components based on the specified performance target (i.e., composition success rate) and current system conditions (e.g., application workload, available resources). For the latter problem, ACP combines distributed composition probing with a hierarchical state management scheme. The hierarchical state manager updates the global state information in a coarse-grained fashion while keeping local state information precise using frequent update. Thus, ACP can intelligently select a subset of good candidate components to probe under the guidance of the coarse-grained global state information. The best component composition will be selected based on the precise states collected by the probes.

Compared to previous centralized approaches [5, 9], ACP achieves better scalability by replacing precise global state maintenance with coarse-grain global state update and on-demand precise state collection from distributed hosts. Compared to previous fully decentralized approaches [4, 13], ACP achieves better performance by querying the coarse-grain global state. Hence, ACP provides a *hybrid* approach that can achieve both efficiency and scalability in optimal component composition. We have conducted extensive simulation study to experimentally evaluate the ACP approach. Our results show the efficiency, scalability, and adaptability of the ACP approach by comparing with the optimal algorithm that has exponential overhead, and other common heuristic approaches.

The rest of the paper is organized as follows. Section 2 introduces the system model. Section 3 presents the ACP approach in detail. Section 4 presents the experimental results. Section 5 discusses related work. Finally, the paper concludes in Section 6.

## 2 System Model

In this section, we present the distributed stream processing system architecture, stream processing request model, and the formal definition of the optimal component composition problem.

### 2.1 System Architecture

The distributed stream processing system, illustrated by Figure 1 (a), consists of a collection of stream processing nodes ($v_i$), each of which can be a single computer or a computer cluster. For failure resilience, we connect distributed nodes using application-level overlay links ($e_i$) into an overlay mesh. Each node provides a set of stream processing components $\{c_1, .., c_k\}$. Each component provides an atomic stream processing function ($F_i$) such as filtering, aggregation, correlation, and audio/video analysis.

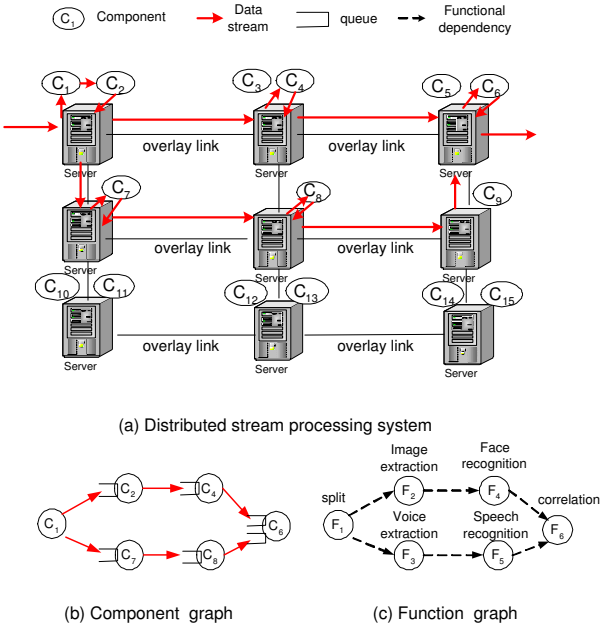



**Figure 1. Stream processing system model.**

Due to the constraints of security, software licence, and hardware requirements, we do not assume that each node can provide all stream processing components. The component composition process is to select and connect currently deployed components into user required stream processing applications[1].

Each component receives continuous data units (e.g., data tuples, audio samples, video frames) via input queues from its preceding components, illustrated by Figure 1 (b). Each component has well-defined interfaces describing its input requirements (e.g., data format, stream rate) and output properties. Each component is associated with (1) a QoS vector $[q_1^{c_i}, ..., q_m^{c_i}]$ describing the component's QoS values such as processing time and loss rate[2], and (2) a resource availability vector $[ra_1^{c_i}, ..., ra_n^{c_i}]$ describing the current available resources (e.g., CPU, memory), on the node providing $c_i$. Similarly, we also associate a QoS vector $[q_1^{e_i}, ..., q_m^{e_i}]$ and bandwidth availability $ba^{e_i}$ with each overlay link $e_i$.

Distributed components can be dynamically composed into composite stream processing applications. Generally, composition topology can be a directed acyclic graph (DAG), illustrated by Figure 1 (b). We use component graph ($\lambda$) to represent a composed stream processing application. Although the input queue can accommodate transient stream rate mismatch, the input/output rates of two adjacent components must be compatible to provide

[1] Components can be dynamically migrated among nodes. The component composition operates based on the current component placement.

[2] The component can drop data units when it is overloaded [12].

continuous processing of long-lived data streams. Such a compatibility check is based on the component's interface specifications. The component's input/output rates are controlled by resource allocation policies. The QoS values of a composed stream processing application is the aggregation of QoS values among its constituent components and overlay links[3]. The connection between two adjacent components is called *virtual link* ($l_i$), which consists of a set of overlay links. The QoS of the virtual link $[q_1^{l_i}, ..., q_m^{l_i}]$ is the aggregation of QoS values among its constituent overlay links; The bandwidth availability $ba^{l_i}$ is the bottleneck bandwidth among the overlay links[4] .

## 2.2 Stream Processing Request Model

The user can specify the stream processing request in terms of: (1) function requirements described by a function graph ($\xi$), (2) QoS requirements ($Q^{req}$), and (3) resource requirements ($R^{req}$). The function graph, illustrated by Figure 1 (c), defines a stream processing application template, which can be provided by the application developer. The function graph includes a set of function nodes ($F_i$) connected by dependency links. Different from the conventional request-response middleware interface, our stream processing middleware provides session-oriented interfaces:

- `sessionId` = `Find` ($\xi$, $Q^{req}$, $R^{req}$) invokes the optimal component composition algorithm to find the best component graph $\lambda$. If the composition is successful, the middleware creates a session record with a session identifier (sessionId) for the user request. Otherwise, a null sessionId is returned to indicate composition failure.
- `Process`(`sessionId`, `data streams`) starts the continuous data stream processing using the application's component graph. The middleware can map the session identifier to the component graph composed by the previous step.
- `Close`(`sessionId`) tears down the stream processing session when the application finishes its task. The corresponding session information is deleted from the session table.

## 2.3 Problem Description

The problem of optimal component composition is to compose best stream processing applications using existing components in the distributed stream processing system. The optimal component composition has two principle goals. First, the composed stream processing application should satisfy the user's function, QoS, and resource requirements. Second, the stream processing application should be instantiated on least loaded nodes and virtual links for balanced load distribution. We use $[rr_1^{c_i}, ..., rr_n^{c_i}]$ to define the *residual* end-system resources on the node providing $c_i$, and $rb^{l_i}$ to define the *residual* bandwidth on the virtual link $l_i$, respectively. The residual resources are the *remaining* resources *after* the required resources are subtracted from current available resources on the corresponding nodes and virtual links[5]. We use $R^{c_i} = [r_1^{c_i}, ..., r_n^{c_i}]$ to define the required end-system resources by the component $c_i$. The bandwidth requirement of the virtual link $l_i$ is denoted by $b^{l_i}$. We use $c_i.f$ to denote the stream processing function provided by the component $c_i$. Thus, the optimal component composition problem can be formally defined as follows,

**Definition 2.1.** *Given a distributed stream processing system G = (V,E) where $V$ denotes the set of $|V|$ stream processing nodes ($v_i$) and $E$ denotes the set of $|E|$ overlay links ($e_i$). Given a stream processing request $\langle \xi, Q^{req}, R^{req} \rangle$ where $\xi$ denotes the function graph, $Q^{req}$ denotes the QoS requirements, and $R^{req}$ denotes the resource requirements. The* **optimal component composition** *problem is to find the best component graph $\lambda = (C, L) \subset G$, where $C$ denotes the set of $|C|$ stream processing components ($c_i$) and $L$ denotes the set of $|L|$ inter-component virtual links ($l_i$) such that*

$$\textbf{min } \phi(\lambda) = \sum_{c_i \in \lambda} \sum_{k=1}^{n} \frac{{r_k}^{c_i}}{{rr_k}^{c_i} + {r_k}^{c_i}} + \sum_{l_i \in \lambda} \frac{b^{l_i}}{rb^{l_i} + b^{l_i}} \quad (1)$$

$$\textbf{subject to } c_i.f = F_i, \ \forall F_i \in \xi, \exists c_i \in \lambda \quad (2)$$

$$q_i^{\lambda} \leq q_i^{req}, \ \ 1 \leq i \leq m \quad (3)$$

$$rr_k^{c_i} \geq 0, \ \ 1 \leq k \leq n, \forall c_i \in \lambda \quad (4)$$

$$rb^{l_i} \geq 0, \ \ \forall l_i \in \lambda \quad (5)$$

Generally speaking, the optimal component composition problem is a multi-constrained optimization problem, which optimizes a global system metric subject to a set of constraints. In this paper, the optimization goal is to minimize the *congestion aggregation* metric $\phi(\lambda)$ defined by equation 1 for balanced load distribution. The smaller the congestion aggregation metric is, the better load distribution the composition presents since we instantiate the stream processing application on the nodes and virtual links with larger residual resources. The constraints include the user's function, QoS and resource constraints for the composed

[3]In this paper, we assume that QoS metrics are additive and minimum-optimal. For non-additive metrics (e.g., loss rate), we can make them additive and minimum-optimal using logarithm and inverse transformations.

[4]If two adjacent components are co-located on the same machine, the virtual link is said to have 0 network delay and $\infty$ bandwidth availability.

[5]We use residual resources instead of current available resources for accommodating component co-locations, which will be explained in Section 3 with more details.

stream processing application. Equation 2 defines that the component graph $\lambda$ must provide all stream processing functions specified in the function graph $\xi$. Equation 3 specifies that the QoS of the composed stream processing application $[q_1^\lambda, ..., q_m^\lambda]$ (e.g., processing time, loss rate) must satisfy the user QoS requirements $[q_1^{req}, ..., q_m^{req}]$. Equation 4 and equation 5 specify that user required system resources and bandwidth resource must be satisfied (i.e., residual resources cannot be negatives).

We can prove that the optimal component composition problem is NP-hard since the *multi-constrained path selection* problem, which is known to be NP-hard [3], maps to a special case of the optimal component composition problem. Besides its computation intractability, it is also challenging to acquire up-to-date QoS/resource states required by the optimal component composition in a large-scale distributed stream processing system. Thus, the goal of our solution is to provide an efficient, scalable and self-tuning approximation solution for the optimal component composition problem.

# 3 Design and Algorithm

In this section, we present the adaptive composition probing (ACP) approach in detail. We describe the ACP overview, the hierarchical state management, the ACP algorithm, the probing ratio tuning scheme, and the hop-by-hop component selection algorithm.

## 3.1 Approach Overview

The basic idea of the ACP approach is to use a number of probing messages, called probes, to dynamically discover a set of good candidate compositions among which the best composition is selected. The probing process concurrently examines different compositions and collects precise state information from good candidate components. For scalability, ACP avoids brute-force exhaustive probing by performing adaptive selective probing. ACP addresses two key decision-making problems in composition probing: (1) *how many candidate components should we probe?* and (2) *which candidate components should we probe?*

Intuitively, the more candidate components we probe, the better component composition we can discover. However, the probing overhead also becomes larger when we probe more candidate components. We introduce *probing ratio* $\alpha \in (0, 1]$ to define the percentage of the candidate components we probe for each function. For example, if there are ten candidate components for the function $F_i$ and the probing ratio $\alpha = 0.3$, then we can probe $0.3 \times 10 = 3$ candidate components. ACP adaptively adjusts the probing ratio based on the target composition performance

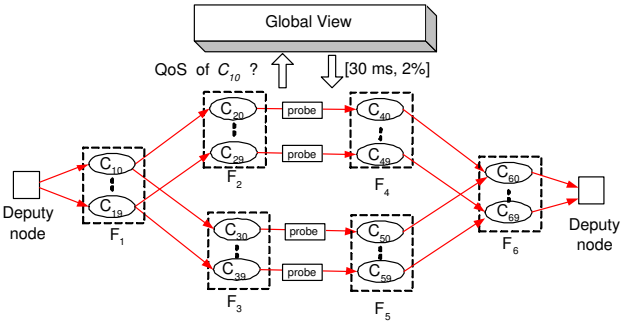



**Figure 2. Adaptive composition probing.**

and current system conditions, which will be described in Section 3.4 in detail.

Next, we need to decide which candidate components to probe for maximizing the probability of finding the best composition. We propose a hierarchical state management scheme to assist ACP in optimal component composition. The hierarchical state manager maintains precise local state at each node and a coarse-grain global state, which will be descried in Section 3.2. Hence, ACP can select good components to probe under the guidance of the coarse-grain global state and select optimal component composition based on precise states collected by the probes. The details about the candidate component selection will be presented in Section 3.5.

## 3.2 Hierarchical State Management

The hierarchical state management consists of *fine-grain* local state update and *coarse-grain* global state maintenance. The local state of a node consists of the QoS/resource states of its neighbor nodes in the overlay mesh, and its adjacent overlay links. Each node keeps its local state with high precision using frequent proactive measurement at short time interval (e.g., 10 seconds). For scalability, the precise local state is not disseminated to other nodes.

The global state consists of: (1) the QoS and resource states of all nodes, and (2) the QoS and resource states of all virtual links between all pairs of nodes. Since each node can provide multiple stream processing components, the QoS states of each node include the QoS states of all stream processing components it provides. For scalability, the global state update is performed at a coarse-grain level. The global state update is triggered only when state variations on a node or an overlay link exceed a specified threshold. Thus, many insignificant state variations are filtered out to reduce the global state maintenance overhead. For example, a node updates its available memory state in the global state only when the available memory variation is larger than 100

KB. Similarly, a node updates the available bandwidth of its adjacent overlay link in the global state only when the available bandwidth variation is more than 200 kbps.

One complication in the global state update is that each virtual link is actually an overlay path consisting of several overlay links. Thus, the virtual link state update requires further aggregation from the states of its constituent overlay links. For example, we want to update the available bandwidth of a virtual link ($l_i$) in the global state, which is mapped to an overlay path $l_i = \langle e_1, ..., e_k \rangle$. Then, we need to calculate the available bandwidth of $l_i$ from the available bandwidth of its constituent overlay links: $ba^{l_i} = min(ba^{e_i}, ..., ba^{e_k})$. Similarly, we need to calculate QoS values (e.g., delay, loss rate) for the virtual link $l_i$ before we can update its state in the global state. Thus, we select one node (e.g., the least loaded node) as the aggregation node to calculate the states of all virtual links. All other nodes send *significant* QoS/resource state variations of their adjacent overlay links to the aggregation node. The aggregation node periodically updates the global state with the states of all virtual links between all pairs of nodes in the overlay mesh at large time interval (e.g., 10 minutes). For load sharing, we switch the aggregation role among all system nodes (e.g., round robin or least loaded first).

### 3.3 The ACP Algorithm

When a stream processing request is submitted, the request is redirected to a node that is closest to the client based on a predefined proximity metric (e.g., geographical location). The selected node, called *deputy node*, initiates the ACP protocol to discover the optimal component composition. Figure 3 shows the pseudo-code of the ACP protocol that mainly includes the following steps:

**Step 1. Initialization.** Given the component composition request, the deputy node first selects a proper probing ratio for the request, which will be described in Section 3.4. The probing ratio decides the number of candidate components we probe for each required function. Next, the deputy node creates a probing message ($P_0$) that carries the composition request information (e.g., the function graph $\xi$, QoS constraints $Q^{req}$, resource constraints $R^{req}$) and the probing ratio $\alpha$. The ACP protocol then enters the next step: distributed hop-by-hop probe processing. The probes will visit a number of selected candidate components to collect precise QoS/resource states.

**Step 2. Per-hop probe processing.** When a node[6] $v_i$ receives a probe $P_i$, it processes the probe independently based on the information carried by the probe, the local state, and the coarse-grain global state. First, $v_i$ checks whether the QoS/resource values of the probed partial stream processing application already violate the required

[6]This step applies to the deputy node too.

**Input:** Request $\langle \xi, Q^{req}, R^{req} \rangle$, deputy node $v_0$;
**Output:** best component composition $\lambda$.
1. $v_0$ creates the initial probe $P_0$
   Calculate probing ratio $\alpha$
   Create initial probe $P_0 = (\xi, Q^{req}, R^{req}, \alpha)$
2. Per-hop probe processing at node $v_i$
   $v_i$ checks resource/QoS conformance
   **if** the probed composition is qualified **then**
      Perform transient resource allocation
      Derive next-hop functions using $\xi$
      Discover next-hop candidate components
      Select good candidates using global state
      Spawn probes for selected components
      Update probes with fine-grain local states
      Send probes to their next-hop components
   **else** Drop the received probe P
3. $v_0$ selects the best component composition
4. $v_0$ establishes the stream processing session

**Figure 3. The ACP algorithm.**

QoS/resource values. If the QoS/resource values are already unqualified, the probe is dropped immediately to reduce the probing overhead. Otherwise, the node performs *transient* resource allocation to avoid conflicting resource admission caused by concurrent probings for different requests[7]. The transient resource allocation will be cancelled after a time-out period if the node does not receive a confirmation message. Second, $v_i$ derives the next-hop functions using the function graph $\xi$. The functions dependent on the current function are the next-hop functions. Third, $v_i$ acquires the locations of all available candidate components for each next-hop function using a decentralized service discovery system [6]. Fourth, $v_i$ selects a number of *good* candidate components to probe, based on the probing ratio and the global state. The details about the candidate component selection will be described in Section 3.5. Fifth, $v_i$ *spawns* new probes from $P_i$ for all selected next-hop candidate components. Each new probe collects fine-grain local states from $v_i$ and inherits the states collected by its parent probe. Finally, $v_i$ sends all new probes to the selected next-hop components.

**Step 3. Optimal composition selection.** All the probes belonging to a composition request will return to the initial deputy node. First, if the function graph is a DAG, the deputy node derives complete component graphs by merging the probed component paths. For example, in Figure 2, a probe can traverse $c_{10} \rightarrow c_{20} \rightarrow c_{40} \rightarrow c_{60}$ or $c_{10} \rightarrow c_{30} \rightarrow c_{50} \rightarrow c_{60}$. We can then merge these two component paths into a complete component graph. Second, the deputy node calculates the residual resources and accumulated QoS

[7]To avoid redundant resource allocation, each node only temporarily reserves resources *once* for each component in each request.

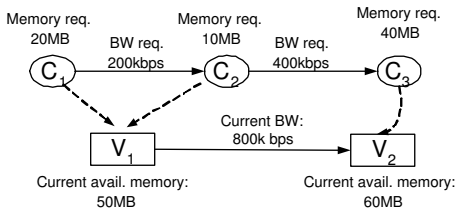



**Figure 4. Congestion aggregation calculation.**

values for the candidate component graphs based on the precise states collected by the probes. Third, the deputy node selects qualified component compositions according to the constraints specified by Equations 2-5. Fourth, the deputy node selects the best component composition from all the qualified compositions according to the congestion aggregation metric $\phi(\lambda)$ defined by Equation 1. For example, in Figure 4, we calculate the residual resource aggregation for the candidate component composition as[8]: $\phi(\lambda) = \frac{20}{20+20} + \frac{10}{20+10} + \frac{40}{20+40} + \frac{200}{\infty+200} + \frac{400}{400+400} = 2$. The qualified composition with the smallest $\phi(\lambda)$ value is the optimal component composition.

**Step 4. Application session setup.** Finally, the deputy node establishes the stream processing application session by sending confirmation messages along the selected component composition. The confirmation message makes transient resource allocation permanent on the selected nodes and virtual links. If no qualified component composition can be found, the deputy node returns a failure message.

### 3.4 Probing Ratio Tuning

We now describe the probing ratio tuning scheme. If a function $F_i$ has $k_i$ candidate components and the probing ratio is $\alpha$, ACP will probe $\lceil \alpha \cdot k_i \rceil$ candidate components for the function $F_i$. Intuitively, the larger the probing ratio is, the better composition performance the ACP can achieve since more candidate components are examined. However, larger probing ratio also means larger probing overhead since more probes are generated. Thus, the probing ratio represents a tuning knob to control the trade-off between composition performance and probing overhead. We use *composition success rate* $\mu(t)$ to characterize the system's composition performance at sampling time $t$. The composition success rate is calculated by $\mu(t) = \frac{SuccessNum(t)}{RequestNum(t)}$, where $SuccessNum(t)$ denotes the number of successful compositions during last sampling period $[t - \triangle t, t]$ and $RequestNum(t)$ denotes the number of total composition requests during $[t - \triangle t, t]$. Higher composition success

[8]If two components are located on the same node (e.g., $c_1$ and $c_2$), the residual bandwidth between the two components is set $\infty$ since the virtual link between two co-located components do not consume any network bandwidth.

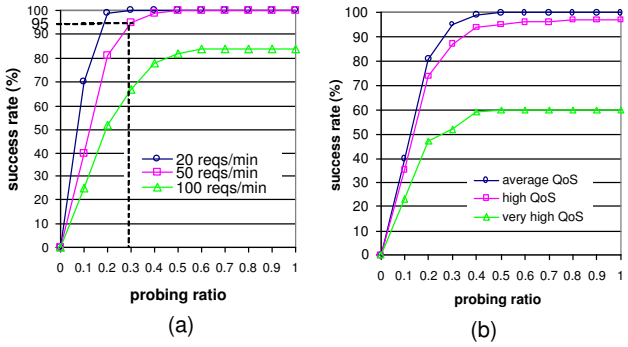



**Figure 5. Probing ratio tuning effect.**

rate $\mu(t)$ indicates better optimal component composition performance at time $t$. We use $\alpha(t)$ to denote the probing ratio value at time $t$.

Ideally, ACP should always use the *minimal* probing ratio $\alpha(t)$ for achieving the target composition success rate $\mu(t)$ independent of system conditions (e.g., workload changes, system node changes). For example, Figure 5 shows the composition performance (i.e., success rate) as the function of probing ratio under increasing workload and higher QoS requirements. We use request rate to denote the number of requests submitted per minute. Higher request rate represents larger workload. For example, in Figure 5 (a), if we want to achieve $\mu(t) = 95\%$ composition success rate when the request rate is 50 requests per minute, ACP should set the probing ratio $\alpha(t) = 0.3$. However, the challenge of probing ratio tuning is that the mapping from probing ratio to success rate is non-linear and dynamic. The mapping function can be affected by different system conditions (e.g., request rate, QoS/resource requirements) in a dynamic distributed stream processing system. For example, Figure 5 (a) shows the mapping functions from probing ratio to composition success rate under different request rate. Figure 5 (b) shows the composition success rate as the function of the probing ratio under different QoS requirements. Higher QoS means shorter processing time and lower loss rate requirements.

To address the problem, ACP performs on-line profiling to dynamically derive the mapping function from probing ratio to composition success rate. Based on the profiling results, ACP can predict the minimal probing ratio $\alpha(t)$ given a target success rate[9] $\mu(t)$. The on-line profiling is triggered when the prediction error exceeds a certain threshold $\delta$ (e.g., $\delta = 2\%$), which means that the system conditions have changed. For example, if the target success rate is $\mu(t)$, ACP predicts the minimal probing ratio $\alpha(t)$

[9]We assume that the target success rate is achievable. ACP stops increasing the probing ratio if the probing overhead already reaches its limit.

based on the current profiling results. At the end of this sampling period, ACP gets the measured success rate $\mu'(t)$. If $|\mu(t) - \mu'(t)| > \delta$, profiling is triggered to derive the new mapping function from probing ratio to composition success rate.

Because ACP is highly efficient, the success rate increases very fast as we increase the probing ratio. The success rate can quickly reaches the saturation point (i.e., the highest achievable success rate) at a small probing ratio, illustrated by Figure 5. Thus, the probing ratio tuning space is very limited, which makes probing ratio profiling a simple task. The profiling process starts from the base probing ratio (e.g., $\alpha = 0.1$) and gradually increases the probing ratio at a certain step (e.g., $0.1$) until the success rate hits the saturation value. To guarantee high prediction accuracy, profiling should use realistic workload that are representative of the current system conditions (e.g., request rate, QoS requirements, resource requirements, application templates). Such a workload can be the trace replay of actual workloads in the last sampling period.

### 3.5 Candidate Component Selection

After the deputy node decides the probing ratio, it executes the ACP protocol to send out composition probes. When a node $v_i$ receives a probe $P_i$, it needs to decide *which* next-hop candidate components to examine under the probing ratio constraint. For example, if the next-hop function $F_i$ has $k_i$ candidate components and the probing ratio is $\alpha$, then the node $v_i$ is allowed to probe $M = \lceil \alpha \cdot k_i \rceil$ candidate components for $F_i$. Under the guidance of the coarse-grain global state, $v_i$ selects $M$ best candidate components as follows:

**First**, $v_i$ queries the global state to retrieve the coarse-grain QoS/resource states of all candidate components $c_1, ..., c_k$. We use $l_i$ to denote the virtual link from the current component to $c_i$. The QoS states of the candidate component $c_i$ and the virtual link $l_i$ are described by $[q_1^{c_i}, ..., q_m^{c_i}]$ and $[q_1^{l_i}, ..., q_m^{l_i}]$, respectively. The resource states of the candidate component $c_i$ include current resource availabilities $[ra_1^{c_i}, ..., ra_n^{c_i}]$. The resource states of the virtual link $l_i$ include its current available bandwidth $ba^{l_i}$. On the other hand, $v_i$ acquires the resource/QoS requirement information from the received probe. We use $[r_1^{c_i}, ..., r_n^{c_i}]$ to denote the application's end-system resource requirements, and $b^{l_i}$ to denote the user's bandwidth requirement. We can calculate the residual resources of $c_i$ as $rr_i^{c_i} = ra_i^{c_i} - r_i^{c_i}$ and the residual bandwidth of $l_i$ as $rb^{l_i} = ba^{l_i} - b^{l_i}$.

**Second**, $v_i$ filters out unqualified candidate components by checking the input/output stream rate compatibility between the current-hop component and candidate next-hop components. Then, $v_i$ further removes unqualified candidate components according to the user's QoS/resource requirements and the state information retrieved from the global state. We use $[q_1^{req}, ..., q_m^{req}]$ to denote the user's QoS requirements for the composed stream processing application. Let us assume that the accumulated QoS values of the partial component composition traversed by the received probe $P_i$ are $[q_1^{\lambda}, ..., q_m^{\lambda}]$. A candidate component $c_i$ is unqualified if any of the following inequalities is true:

$$q_i^{\lambda} + q_i^{c_i} + q_i^{l_i} > q_i^{req}, \exists i, 1 \leq i \leq m \quad (6)$$

$$ra_i^{c_i} < r_i^{c_i}, \exists i, 1 \leq i \leq n \quad (7)$$

$$ba^{l_i} < b^{l_i} \quad (8)$$

The equation 6 means that the user's QoS constraints cannot be satisfied. The equation 7 means that the candidate component $c_i$ cannot meet the end-system resource requirements. The equation 8 means the virtual link to $c_i$ cannot meet the bandwidth requirement.

**Third**, $v_i$ further selects *good* candidate components from the above derived *qualified* components. Let us assume $v_i$ finds $Z$ qualified candidate components. If $Z \leq M$, then $v_i$ can probe all the qualified candidate components satisfying the probing ratio constraint. Otherwise, $v_i$ needs to select $M$ best qualified components from $Z$ qualified ones. To meet the multi-constrained QoS requirements $[q_1^{req}, ..., q_m^{req}]$, we define a *risk function* $D(c_i)$ for a candidate component $c_i$ as follows,

$$D(c_i) = \max\left(\frac{q_1^{\lambda} + q_1^{c_i} + q_1^{l_i}}{q_1^{req}}, ..., \frac{q_m^{\lambda} + q_m^{c_i} + q_m^{l_i}}{q_m^{req}}\right) \quad (9)$$

The larger the ratio $\frac{q_i^{\lambda} + q_i^{c_i} + q_i^{l_i}}{q_i^{req}}$ is, the more closely the QoS accumulation $(q_i^{\lambda} + q_i^{c_i} + q_i^{l_i})$ approaches its constraint $q_i^{req}$. The candidate components with smaller risk function values are considered as better components since their maximum QoS violation risk values are smaller. Thus, $c_i$ is a better candidate component to probe than $c_j$ if $D(c_i) < D(c_j)$. If two candidate components have similar risk function values, we compare them based on the load distribution goal. We define a congestion function $W(c_i)$ as follows,

$$W(c_i) = \sum_{k=1}^{n} \frac{r_k^{c_i}}{{rr_k}^{c_i} + r_k^{c_i}} + \frac{b^{l_i}}{rb^{l_i} + b^{l_i}} \quad (10)$$

The component with smaller $W(c_i)$ values is a better candidate to probe since it is less loaded. The above candidate component selection scheme guarantee that probes traverse along *good* candidate compositions. The fine-grain QoS/resource states collected by the probes will be used to select the best component composition. Thus, ACP can most efficiently find the optimal component composition although it only examines a subset of all candidate compositions.

## 4 Experiments

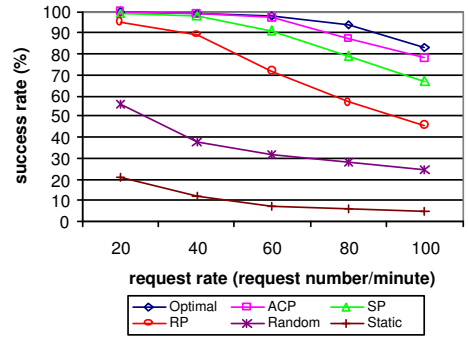
We have conducted extensive simulation study to experimentally evaluate the ACP approach[10]. Our results show that ACP can achieve better efficiency, scalability, and adaptability than other common approaches.

### 4.1 Evaluation Methodology

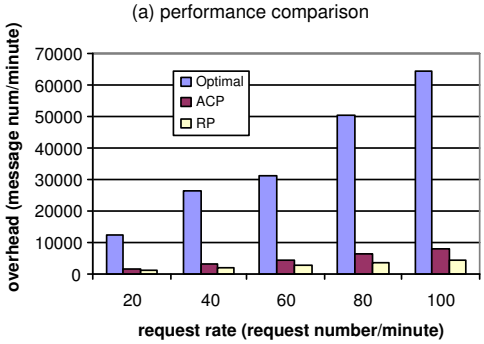
We implemented an event-driven optimal component composition simulator in C++. The simulator first uses the degree-based Internet topology generator Inet-3.0 [14] to generate a 3200 node power-law graph to represent the IP-layer network. The simulator then randomly selects $N \in [200, 500]$ nodes as stream processing nodes, which are connected into an overlay mesh. Each node has $10\% \cdot N$ neighbors. The initial resource capacities and QoS states of stream processing nodes and network links are uniformly distributed within certain range based on the real-world measurements. The simulator simulates both IP-layer and overlay data routing using delay-based shortest path routing algorithm. Each node provides a number of components whose functions are selected from 80 pre-defined functions. The function graph of a stream processing request is randomly selected from 20 pre-defined stream processing application templates. Each function graph is either a path or a DAG with two branch paths. Each path or branch path includes [2,5] nodes. The resource and QoS requirements are uniformly distributed. Each application session lasts 5 to 15 minutes. The global state update is triggered when the value variation of a resource or QoS metric exceeds $10\%$ of its maximum value.

For comparison, we also implement three other common approaches: *optimal*, *random*, and *static* algorithms. The optimal algorithm exhaustively searches all candidate component compositions to find the best composition. The random algorithm randomly selects a candidate component for each required function. The static algorithm selects a fixed candidate component for each function. We also implement two other *composition probing* approaches: *selective probing* (SP) and *random probing* (RP). The SP approach only uses the ACP's per-hop candidate component selection scheme but replaces the optimal composition selection (Equation 1) with random composition selection. The RP approach performs random per-hop candidate component selection but uses the ACP's optimal composition selection scheme. The RP approach represents the fully distributed approach since it only requires local states. Both SP and RP approaches use the same probing ratio as the ACP approach.

[10]We have implemented a prototype of distributed component composition system and deployed the system in the wide-area network testbed PlanetLab. The prototype implements bounded composition probing (i.e., a simpler version of ACP) and supports multimedia stream processing. Readers are referred to [4] for more details. Integrating the ACP scheme into the prototype is one of our on-going work.




(a) performance comparison




(b) overhead comparison

**Figure 6. Efficiency evaluation.**

### 4.2 Results and Analysis

First, we evaluate the efficiency of the ACP approach illustrated by Figure 6. In this set of experiments, we use a 400-node distributed stream processing system and a fixed probing ratio $\alpha = 0.3$. In Figure 6 (a), the x-axis shows different request rate; The y-axis shows the average success rate achieved by different algorithms measured over all the requests generated during 100-minute simulation. We observe that ACP consistently achieves better performance than other heuristic algorithms and similar performance as the optimal algorithm. Figure 6 (b) shows the overhead of different algorithms. The optimal algorithm's overhead is measured by the number of probes required by the exhaustive search per minute. The ACP's overhead is measured by the number of probes and global-state update messages generated per time minute. The RP's overhead only includes the probing overhead. Compared to the optimal algorithm that uses brute-force exhaustive search, ACP can reduce overhead by as much as 95%. Compared to the RP approach, ACP can achieve much better performance by paying a small coarse-grain global state maintenance

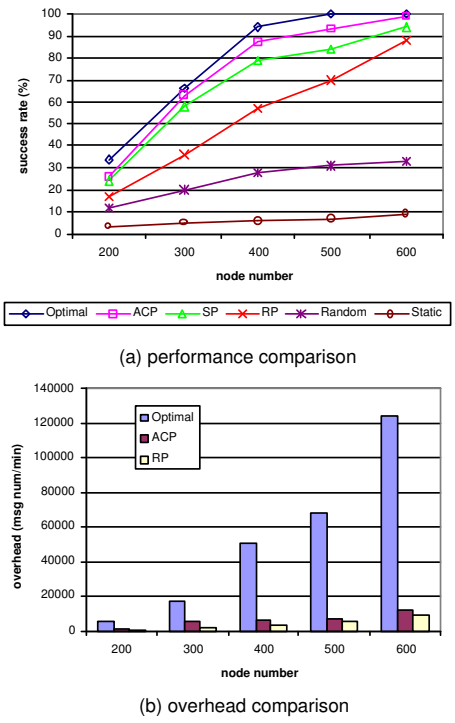



**Figure 7. Scalability evaluation.**

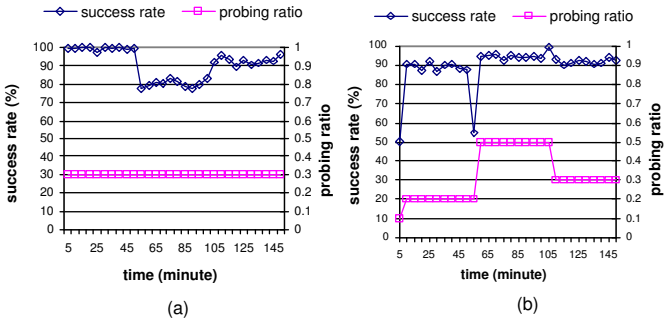



**Figure 8. Adaptability evaluation.**

overhead. In contrast, the centralized algorithm would require $400^2$ messages per minute to perform precise global state update assuming one minute update period is good enough. Hence, ACP can achieve near-optimal performance with similar overhead as the fully distributed approach.

Second, we evaluate the scalability of the ACP approach illustrated by Figure 7. We use different distributed stream processing systems with 200 to 600 nodes. As we add more nodes into the distributed stream processing, the number of candidate components for each function increases proportionally so as to increase the capacity of the distributed stream processing system. We impose the same workload (request rate = 80 requests/minute) on those different distributed stream processing systems. Figure 7 (a) shows the performance comparison results. We observe that ACP achieves similar scaling property as the optimal algorithm. Figure 7 (b) shows that ACP has much lower overhead than the optimal algorithm. The overhead reduction increases as the node number increases.

Finally, we evaluate the adaptability of the ACP approach, illustrated by Figure 8. We impose a dynamic workload on a 400-node distributed stream processing system and set the target composition success rate at $90\%$. The success rate sampling period is $5$ minutes. During a 150-minute simulation, the dynamic workload starts at 40 requests/minute, then increases to 80 requests/minute at time 50, and drops to 60 requests/minute at time 100. Figure 8 (a) shows the performance of the probing approach that uses a *fixed* probing ratio $\alpha = 0.3$. The success rate starts from $100\%$ and drops to about 80% when the request rate increases to 80, and then comes back to about 90% when the request rate decreases to 60. We then enable probing ratio tuning and repeat the same experiment, illustrated by Figure 8 (b). We observe that ACP can effectively adjust the probing ratio to maintain the composition success rate at the target value 90%. The probing ratio is initially $0.1$ and increases to $0.2$ after the first sampling period at time 5. After the request rate increases to 80 at time 50, the success rate drops to 55%, which triggers ACP to increase the probing ratio to 0.5. Finally, the probing ratio drops to 0.3 at time 105 after the request rate decreases to 60 at time 100. The above experimental result demonstrates that ACP can maintain a target composition success rate in a dynamic stream processing environment (e.g., changing workload) by adaptively tuning the probing ratio.

## 5 Related Work

Recently, stream processing has received much research attention. Researchers have proposed load shedding solutions (e.g., [12]) to trade-off processing precision for timely response when the system experiences resource shortage. Other projects have addressed the problems of load balancing [11] and load migration [1] in either cluster-based or wide-area distributed stream processing systems. Grid-based middleware system [2] has also been proposed for distributed stream processing systems. Different from the above work, our research focuses on providing optimal component composition for distributed stream processing systems. The contribution of our research is to provide a stream processing application composition solution that considers load distribution, QoS provisioning, and function dependency.

Component composition has been studied under different research context, such as service composition (e.g.,[9,

13, 5, 6]), systems software composition [10, 7], and multimedia application configuration (e.g., [8]). In contrast, this paper focuses on scalable optimal component composition that is especially important for distributed stream processing systems. Previous work either ignores the composition optimality issue or only addresses part of it. Different from previous centralized resource-aware service composition solutions [5, 9] or fully decentralized approaches [6, 4], ACP represents a *hybrid* approach that uses global state to guide distributed composition probing for better performance. Moreover, ACP can adaptively adjust the probing ratio to achieve target composition performance with minimal probing overhead.

## 6 Conclusion

We have presented the adaptive composition probing (ACP) approach to provide an efficient, scalable, and self-tunable approximation solution for the optimal component composition problem. Our ACP approach can achieve balanced load distribution while satisfying user's function, resource, and QoS constraints in a distributed stream processing environment. Compared to previous approaches, ACP makes two novel contributions. First, ACP proposes a new hybrid approach that combines distributed composition probing with hierarchical state management. Second, ACP is *self-tuning*, which can adaptively adjust the number of probes to maintain target composition performance with minimal probing overhead. Our extensive simulation results show that ACP can achieve better efficiency, scalability, and adaptability compared to other common approaches. Future research directions for optimal component composition include: (1) applying control theory to tune the probing ratio more precisely, (2) supporting other application specific constraints (e.g., security level, software licence) in component composition, and (3) integrating dynamic component placement (or migration) with the component composition system.

## 7 Acknowledgment

We would like to thank Dr. Joel L. Wolf at IBM T.J. Watson research center for his helpful input to our work. We wish to thank anonymous reviewers for their helpful suggestions.

## References

[1] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based Load Management in Federated Distributed Systems. *Proc. of 1st Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA*, March 2004.

[2] L. Chen, K. Reddy, and G. Agrawal. GATES: A Grid-Based Middleware for Distributed Processing of Data Streams. *Proc. of IEEE International Symposium on High-Performance Distributed Computing (HPDC-13), Honolulu, Hawaii*, June 2004.

[3] M. R. Garey and D. S. Johnson. Computers and Intractability. *A Guide to the Theory of NP-Completeness, Freeman, San Francisco*, 1979.

[4] X. Gu and K. Nahrstedt. Distributed Multimedia Service Composition with Statistical QoS Assurances. *IEEE Transactions on Multimedia (to appear)*, 2005.

[5] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. QoS-Assured Service Composition in Managed Service Overlay Networks. *Proc. of IEEE 23nd International Conference on Distributed Computing Systems (ICDCS 2003), Providence, RI*, May 2003.

[6] X. Gu, K. Nahrstedt, and B. Yu. SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. *Proc. of IEEE International Symposium on High-Performance Distributed Computing (HPDC 2004), Honolulu, Hawaii*, June 2004.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems 18(3), pp. 263-297*, August 2000.

[8] R. Koster, A. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for Composing Distributed Information Flows. *Proc. of the ACM Multimedia Workshop on Multimedia Middleware, Ottawa, Canada*, October 2001.

[9] B. Raman and R. H. Katz. Load Balancing and Stability Issues in Algorithms for Service Composition. *Proc. of IEEE INFOCOM 2003, San Francisco, CA*, April 2003.

[10] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. *Proc. of 4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA*, October 2000.

[11] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. *Proc. of the 19th International Conference on Data Engineering (ICDE 2003)*, March 2003.

[12] N. Tatbul, U. etintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. *Proc. of the 29th International Conference on Very Large Data Bases (VLDB'03), Berlin, Germany*, September 2003.

[13] M. Wang, B. Li, and Z. Li. sFlow: Towards Resource-Efficient and Agile Service Federation in Service Overlay Networks. *Proc. of 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan*, March 2004.

[14] J. Winick and S. Jamin. Inet3.0: Internet Topology Generator. *Tech Report UM-CSE-TR-456-02 (http://irl.eecs.umich.edu/jamin/)*, 2002.