FleXR: A System Enabling Flexibly Distributed Extended Reality

Jin Heo
Georgia Institute of Technology
Atlanta, Georgia, USA
jheo33@gatech.edu

Ketan Bhardwaj
Georgia Institute of Technology
Atlanta, Georgia, USA
ketanbj@gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
Atlanta, Georgia, USA
ada@cc.gatech.edu

ABSTRACT

Extended reality (XR) applications require computationally demanding functionalities with low end-to-end latency and high throughput. To enable XR on commodity devices, a number of distributed systems solutions enable offloading of XR workloads on remote servers. However, they make a priori decisions regarding the offloaded functionalities based on assumptions about operating factors, and their benefits are restricted to specific deployment contexts. To realize the benefits of offloading in various distributed environments, we present a distributed stream processing system, FleXR, which is specialized for real-time and interactive workloads and enables flexible distributions of XR functionalities. In building FleXR, we identified and resolved several issues of presenting XR functionalities as distributed pipelines. FleXR provides a framework for flexible distribution of XR pipelines while streamlining development and deployment phases. We evaluate FleXR with three XR use cases in four different distribution scenarios. In the results, the best-case distribution scenario shows up to 50% less end-to-end latency and 3.9× pipeline throughput compared to alternatives.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; **Parallel computing methodologies**; • **Computer systems organization** → *Real-time systems.*

KEYWORDS

distributed stream processing, extended reality, edge computing

ACM Reference Format:
Jin Heo, Ketan Bhardwaj, and Ada Gavrilovska. 2023. FleXR: A System Enabling Flexibly Distributed Extended Reality. In *Proceedings of the 14th ACM Multimedia Systems Conference (MMSys '23), June 7–10, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3587819.3590966

1 INTRODUCTION

Extended reality (XR), including augmented reality (AR) and virtual reality (VR), involves computationally intensive functionalities such as object detection, localization and mapping, and 3D graphics rendering. Given the low-latency and high-throughput requirements of XR applications [42, 56], their associated high computing costs limit the XR experiences that can be supported on resource-constrained mobile devices. To address this, there have been a number of efforts for distributed XR systems [20, 23, 33, 36–41, 44, 52, 66]. A common thread across them is to predetermine the functionalities that are offloaded to the server, based on assumptions about environmental factors: the client device capacities, network conditions, and workloads. They provide support for offloading fixed functional components of perceptions or rendering, and their benefits can be realized only in specific deployment contexts.

However, we argue that the distribution contexts will differ vastly in terms of the capabilities of the user devices and offload servers, the connectivity among them, and the XR workloads. In such scenarios, current solutions will be limited in their ability to provide effective server assistance in various contexts. Applications would need to rely on combinations of existing techniques to leverage the server in assisting with different functionalities. It will require significant additional development and configuration efforts.

Currently, flexibility in XR workload distribution is missing due to a lack of adequate systems support. There are previous offloading systems for flexible function migration [11, 13, 32], but it is hard to extend their benefits to XR due to their design limitation of function-level offloading (see §2 for more details). Existing stream processing (SP) libraries can potentially enable flexible workload distribution by creating a pipeline at runtime. While they are used in use cases similar to XR, *e.g.*, multimedia streaming [21] and perception pipelines [4, 47], the current SP frameworks lack some of the necessary features to adapt distributed stream processing (DSP) for use in XR. As described in §3, a DSP system for XR should support efficient local communication for collocated pipeline components and blocking and non-blocking communication semantics to express the pipeline dependencies and synchronization. Moreover, it should provide queue size management and multiple network protocol supports for data freshness requirements of distributed XR pipelines, which are not available in existing solutions.

Simply adding the missing features to existing SP libraries is not sufficient to enable the flexible distribution of XR pipelines. Even if the SP libraries are extended with those DSP features, there are still issues about how to provide the features properly across the development and deployment phases. Specifically, in existing SP libraries, a pipeline can be created at runtime, and it requires a user to connect pipeline components (compute kernels) via the developer-specified communication ports. However, since the communication attributes among compute kernels are determined under the user's pipeline context, the user (not the developer of the kernel) should configure the communication attributes of the connection ports when creating a distributed pipeline.

In response, we present **FleXR** – an open-source, flexibly configurable, and high-performance system for distributed XR. To bring flexibility, we design FleXR as a DSP system specialized for XR.

MMSys '23, June 7–10, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0148-1/23/06.
https://doi.org/10.1145/3587819.3590966

With FleXR, XR pipelines can be flexibly created for various distribution scenarios at runtime by a user, without requiring any code modification in pipeline kernels. We identify the key issues in using DSP systems for XR (§3.1) and describe our design decisions which provide the necessary DSP features and address them (§3.2). FleXR provides a framework to enable the flexible distribution of XR functionalities, streamlining the development and deployment phases. The developers write their kernels without considering how and where each kernel runs in user pipelines. The user can configure the communication attributes of the given components without any change. This feature is realized by the FleXR's kernel design with its port abstractions and interfaces (§4.2). Once the developer writes a FleXR compute kernel, it can be flexibly deployed and executed in diverse distribution scenarios, per user configuration.

We demonstrate the effectiveness of FleXR through experimental evaluation with three typical XR use cases and four distribution scenarios (§6). Compared to the existing distributed XR systems [9, 20, 33, 36, 37, 52, 66], FleXR is shown to support all distribution scenarios by creating distributed pipelines with given kernels at runtime. Our evaluation results show that the offloading effect of each scenario is different based on the workloads, offloading overheads, and device capacity, which support the importance of flexibility in XR workload distributions. Overall, this paper makes the following contributions:

- We describe the limitations of existing distributed XR systems with respect to the need for flexibility, and identify the required features for applying DSP to XR.
- We present FleXR, a DSP system specialized for distributed XR, which addresses the design issues of DSP for XR and enables flexible distributions of XR pipelines. Our evaluation in different distribution scenarios demonstrates that FleXR practically delivers on the promise of flexibility and performance.
- We fully open-source FleXR, hoping that it would reduce the barriers for further research in the area of distributed XR[1].

2 RELATED WORK AND MOTIVATION

Previous Work and Limitations. Flexibility in XR workload distribution is not currently supported despite extensive prior research and commercial solutions to offload XR functionalities on remote servers. Table 1 summarizes a number of the existing technologies and what they support to be offloaded. For VR, graphics operations are usually offloaded for providing realistic experiences via high-quality rendering. Some studies make use of the characteristics of linear perspective in 3D graphics and split the foreground and background rendering to reuse a hardly changing background [33, 39]. For AR, full and partial offloading of perception modules and graphics rendering have been explored [20, 36, 52]. However, their benefits can be effective only in specific deployment contexts with their static workload distributions.

Flexibility is necessary because the complexities of XR workloads are not the same for each use case, *e.g.*, AR or VR, and vary based on concrete algorithms and applications. Figure 1 shows the normalized execution time for the three different AR and VR applications used later in our evaluation. They are chosen to represent

[1]https://github.com/gt-flexr/FleXR

Table 1: Server-side workloads that can be offloaded with the distributed architecture of existing technologies.

Technologies \ Server-side Workload	Full Offloading	Perceptions	Application Rendering
Marvel [9]		✓	
Glimpse [10]		✓	
OpenRiST [20]	✓		
ISAR [23]	✓		
Furion [33]			✓
Liu *et al.* [36]		✓	
Liu *et al.* [37]			✓
FireFly [38]			✓
Azure Custom Vision [40]		✓	
Azure Remote Rendering [41]			✓
Nvidia CloudXR™ [44]			✓
Schneider *et al.* [52]	✓		
Zhang *et al.* [66]			✓

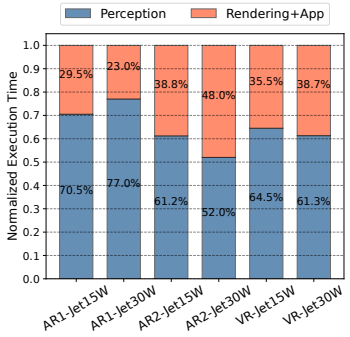


Figure 1: The normalized execution time of our AR and VR examples in different device power modes.

XR use cases with different complexities, which incorporate different algorithms and application frameworks (more detail in §6.2 and Figure 5). They run on NVIDIA Jetson AGX [43] with 15W and 30W power modes, which corresponds to the client device in our testbed in §6.1. These results show each use case has different complexities for its functionalities, and that the dominant functionality – rendering or perception (or both) – depends on the workload and the device capacity, making flexibility in offloading an important consideration in distributed XR.

There have been prior works on flexible offloading to a remote server, but they have limited applicability for XR. MAUI [13] proposed fine-grained function offloading with the common language runtime (CLR) of the .NET framework. By having CLR on the client and server and requiring developers to specify offloadable functions in application codes, the functions are executed flexibly between the client and server. CloneCloud [11] and ThinkAir [32] leveraged OS supports to migrate the execution context of threads running application functions to the server's virtual machines (VM); the server VM provides an environment identical to the client, and the thread execution becomes migratable.

Although these techniques offer some flexibility, their benefit for distributed XR is limited. In their design approach, devices interact with offloaded functions via client-server interfaces, and the application execution flow is preserved. Since XR applications require the processing of multimedia data across multiple functionalities, this can introduce multiple network round trips and compromise the benefits of reduced processing time due to offloading. In addition, preserving the execution flow limits the opportunities to achieve task parallelism. ThinkAir [32] provides parallelism by cloning VMs, but allows only for data parallelism. Lastly, since these systems run

the application codes on a server with the same environment as the client, they cannot benefit from hardware resources only available on the server.

Offloading XR functionalities requires additional operations such as data compression and network transmissions. When distributing XR workloads, both overheads from those auxiliary operations and the costs of the XR functionalities should be considered. Flexibility in reconfiguration thus requires not just techniques which provide transparent function offload, but also systems support to properly configure and deploy the auxiliary functionality.

Motivation for FleXR. These observations motivate us to build FleXR based on the stream processing (SP) design. SP structures an application as a pipeline of components and provides modularity and task parallelism to the application. A pipeline component (a compute kernel) is the implementation of a functionality. Kernels are pipelined via data communication ports and executed in parallel with dataflow. The ports are used to transmit the input and output between the connected kernels. This modularity makes SP extensible to distributed stream processing (DSP) in a straightforward manner; kernels can be connected via remote communication ports. Additionally, SP provides an advantage in heterogeneous server environments [49] since kernels can be specialized to utilize available heterogeneous resources such as hardware accelerators.

3 DESIGN CHALLENGES FOR FLEXR

There are intuitive advantages of building an XR system as an SP system. An XR system processes inputs as data streams from device sensors such as cameras, inertial measurement units, *etc.*, and provides output as data streams such as field of view content in VR and graphic overlay in AR. The SP design provides benefits of high throughput with pipeline parallelism and distributed computation with its modularity, but the application of SP in XR presents non-trivial issues. In this section, we present the design space exploration we conducted to address those issues when building FleXR.

3.1 Issues with Stream Processing for XR

We use an example AR pipeline in Figure 2, to articulate the issues when applying SP to an XR system. The application renderer overlays virtual objects on the camera frame based on the result from the object detector. The objects can be manipulated using key inputs, and the AR scene with the overlaid objects is displayed to the end user. Using this pipeline, we discuss the main issues below.

I1: Communication Cost. As shown in Figure 2, there are a number of components in the pipeline across which data transmissions must occur. Those data transfers need to be performed with least latency because high latency lowers the application's responsiveness and causes discrepancies between the real and virtual worlds. However, the SP design increases the end-to-end latency as it requires data movement across the ports of the pipelined kernels [3, 31]. Since XR functionalities process and produce large multimedia data, the overhead of the cross-kernel communications is significant and must be addressed in order to meet the latency constraints of XR.

I2: Communication Semantics. In Figure 2, the downstream kernels, *i.e.*, the renderer, has input dependencies with the upstream kernels. Some of those dependencies are hard, *i.e.*, an input must be received for the downstream kernel to execute, as is the case with the camera and renderer. Other dependencies are soft, meaning an input is not required from the particular upstream kernel, as is the case with the keyboard or detector and renderer. This property has implications on whether the execution of the upstream kernels should be synchronized with the downstream kernels in terms of their invocation frequencies. In short, the type of dependency is specific to the functionalities that are connected, and this semantic information must be expressed by the application and adequately handled by the underlying system via support for appropriate communication semantics.

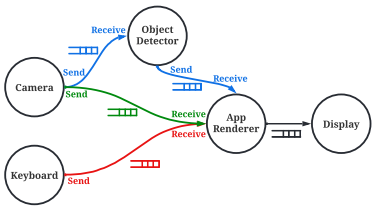


Figure 2: The example pipeline of an AR use case.

Table 2: The local communication latencies between two kernels in milliseconds.

Libraries \ Resolution	720p	1080p	1440p	2160p
ROS Pub/Sub[47]	3.4	6.9	7.1	12.5
ROS Shm Pub/Sub [65]	2.2	4.3	5.9	10.2
Python Queue [12, 60]	14.3	24.1	30.4	52.1
Python Pipe [12, 60]	9.3	17.1	29.5	52.1
Python Shm [17]	3.0	8.6	14.8	32.3
GStreamer [21]	0.1	0.1	0.1	0.1
RaftLib [4]	0.1	0.1	0.1	0.1

I3: Data Recency. If the camera frames in Figure 2 are delayed due to queuing delays when the frame data is transmitted, its freshness decreases. As a result, the placement of the AR object would be off, thus lowering the quality of the AR experience, which is also established by prior work that stale data deteriorates the quality of XR experiences [34]. Generally, as data is transmitted across kernels of different frequencies and execution times, it may result in queuing delays if the data is queued at any of the port buffers in a pipeline. When the data contains real-world contexts from sensors, *e.g.*, camera frames, it is critical to ensure that it remains fresh with all pipeline components that process it. Thus, the SP system for XR must provide a way to manage data recency.

3.2 Design Decisions for The Issues

FleXR as a specialized DSP system for XR, incorporates solutions to address the issues raised in the previous section.

D1: Efficient Local Communication. The remote communication cost is unavoidable even with data compression, but the local communication should be efficient for low overheads. We evaluated the suitability of several existing SP libraries in terms of their communication costs: RaftLib [4], GStreamer [21], Python pipeline libraries [12, 60], and the robot operating system (ROS) [47].

We measure the communication costs with two locally connected kernels and raw RGB frames of different resolutions. Table 2 shows the transmission latencies of the frames. ROS and Python libraries provide process-level SP, where each kernel runs as a separate process and the processes communicate via interprocess communication (IPC) channels. Based on our results, the local communication in process-level SP is hardly efficient for large multimedia data even

with shared memory channels [17, 65]. While the shared memory channel reduces the number of data copies, the data still needs to be copied between the shared memory and process memory regions.

GStreamer and RaftLib provide thread-level SP with zero-copy communication ports. As kernel functions are threads in the same address space, local communication can be done without copy. *A DSP system for XR should leverage a thread-level SP for efficient local communication of collocated kernels, and extend its communication with support for remote communication.*

D2: Blocking and Non-blocking Semantics. To handle the hard and soft dependencies and synchronize the kernel executions in a pipeline, providing the proper communication semantics (blocking and non-blocking) for the local and remote communication primitives (send and receive) is essential [14]. The design of FleXR handles this as a first-order concern when executing XR pipelines.

The send semantics of an output port is for synchronizing the kernel execution. A blocking send blocks the execution of an upstream kernel function when the downstream kernel's queue is full, and this backpressure leads to flow control and implicitly synchronizes the upstream to downstream kernels. For non-blocking semantics, the upstream kernel continues when the downstream kernel cannot receive data on its input port. The output port requires both blocking and non-blocking send semantics in XR pipelines. In the example AR pipeline of Figure 2, if only blocking semantics are supported, the camera kernel is synchronized to the longest path of object detection (blue line). Even if the app renderer does not require the results from the object detector for every camera frame, the frame stream (green line) is blocked by the object detector.

The receive semantics of an input port is for kernel dependencies. A blocking receive waits for the message from a port, and a non-blocking continues when there is no message. So, when the kernel is written, the primary inputs on which the kernel depends (*e.g.*, camera frame for a kernel performing frame processing) should be specified with blocking semantics. For inputs generated by other sources (*e.g.*, other sensors or user events), which can impact or steer the kernel processing but are only optionally used, the semantics should be with non-blocking to handle them.

When only a blocking receive is available, all input streams of a kernel are forced as mandatory. The kernel execution and pipeline throughput are restricted by the lowest frequency input. In Figure 2, the renderer is blocked until the key input arrives from the user (red line). Even without the key input, the renderer execution is governed by the object detector, and the pipeline throughput is limited by the path with the highest latency (blue line).

Supporting both blocking and non-blocking primitives for the input and output ports makes it possible to correctly describe stream dependencies and synchronize kernel executions in XR pipelines.

D3: Queuing Management and Network Protocols. Since poor data freshness causes discrepancies between the real and virtual worlds, it is crucial to manage data recency in XR pipelines. This can be achieved by minimizing the queuing delays of a pipeline [35].

For local communication, it is possible to bound the queuing delay by limiting the number of outstanding data entries in the port buffer. For remote communication, recency management becomes challenging because there is no way to control the queuing mechanisms of unknown middleboxes across the backend network. In this situation, recency management can be enabled by compromising communication reliability. Reliable network protocols such as TCP [55] guarantee in-order message delivery via retransmission and acknowledgment mechanisms. However, in cases where recent data is prioritized (*e.g.*, the object detection result on a live camera frame), the reliable protocols are inappropriate, and should be replaced with protocols favoring data timeliness over reliability even with data loss, *e.g.*, RTP [53] and RTSP [54] over UDP.

Thus, *the DSP system for XR should provide knobs for data recency management via queue size management and support for multiple network protocols for local and remote kernel communications.*

4 FLEXR

4.1 Overview

To bring flexibility to XR workload distribution, we built FleXR as a DSP system specialized for XR, taking the design benefits of modularity and task parallelism. Driven by our design decisions, FleXR is built on top of a thread-level SP library, RaftLib [4], and provides the benefit of efficient local communication for the collocated kernels (**D1**). For the communication semantics to enable kernel synchronization and dependencies of XR pipelines, we extend the semantics of the RaftLib port with support for non-blocking and for remote communication (**D2**). For recency management of local and remote communication, FleXR allows setting the maximum number of messages in the local port buffer and specifying network protocols for remote ports at runtime (**D3**).

Even with the necessary DSP features for XR, there are still issues about providing these features properly to the system stakeholders: developers writing kernels and users requesting distributed pipelines with given kernels. FleXR enables flexibility in configuring XR pipelines via its kernel abstraction with interfaces separating development- and deployment-time concerns. While the developer implements an XR kernel function and knows its input dependencies, the user creates a distributed XR pipeline and configures the connectivity of kernels (local or remote), output-port semantics, and recency management mechanism of the pipeline context. In addition, there can be a case where the user needs to connect an output of a kernel into multiple downstream kernels.

Our kernel abstraction provides the interfaces allowing the developer to register input and output ports and use the registered ports in the kernel function regardless of how they will be configured by a user. The behavior of the registered ports becomes different based on the user-specified communication attributes at runtime. The user can also branch dynamically an output port with different communication attributes and flexibly create distributed pipelines with various topologies without modifying the kernels.

Figure 3 shows the high-level design of FleXR and how it operates. ❶ The developers write kernels with our kernel abstractions for implementing their XR functionalities or incorporating existing functionality implementations by wrapping them in kernel functions. ❷ With given kernels, the user requests a distributed pipeline as a YAML recipe describing pipelined kernels and their communication attributes. ❸ The recipe parser parses the recipe and generates pipeline metadata of the kernels and connection information. ❹ The local pipeline metadata is passed to the pipeline manager, and it creates a local pipeline by instantiating the kernels

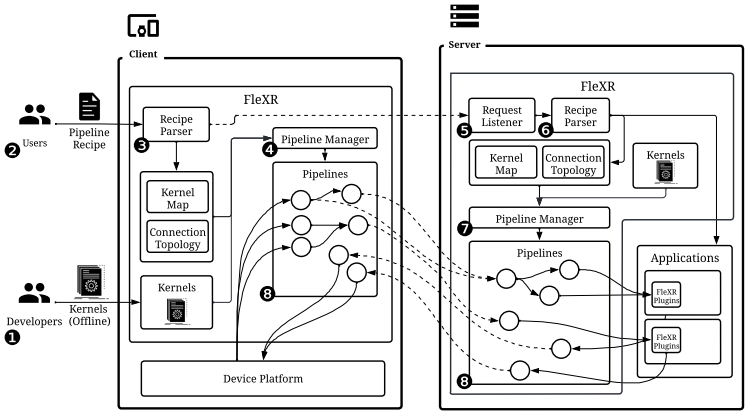


Figure 3: High-level overview of FleXR.

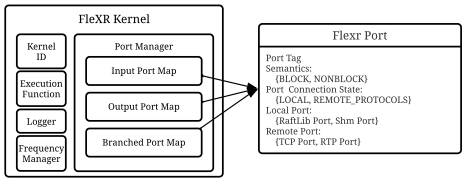


Figure 4: FleXR kernel design with port abstractions.

and configuring port connections. ❺ The part of the recipe about the remote pipeline is sent to the request listener on the server. ❻ The server's recipe parser parses the received recipe and generates pipeline metadata. If the pipeline works with external applications, it starts the applications. ❼ Then, the server's pipeline manager also creates the server-side pipeline, and the remote ports of local and remote pipelines are connected. ❽ The pipelines, distributed across the client and server, run with dataflow.

4.2 Kernel Design

In SP, the compute kernel is a pipeline component, which includes an execution function and communication ports. To provide the necessary DSP features for flexible configuration of the communication attributes, we design a FleXR kernel with two abstractions: the FleXR port and port manager. They abstract the different communication channels for local and remote operation of a FleXR kernel, allowing the developer to write kernels without specifying the communication attributes and the user to configure the connection at runtime without modifying kernels. A developer registers the input and output ports of the kernel with tags via the port manager interface, and the registered ports are instantiated and configured by the port manager based on the user recipe.

Figure 4 shows our kernel design. Each kernel has its ID, logger, frequency manager, execution function, and port manager. The ID is used for the recipe parser and pipeline manager in Figure 3. The frequency manager adjusts the execution frequency when a kernel should run at a stable frequency, and the logger is for the developer to log the kernel events. The execution function processes data from the input ports and sends out the result to the output port.

Port Manager. When kernels are instantiated, the port manager of each kernel activates and dynamically branches the FleXR ports based on the pipeline metadata. In addition, it provides developers with interfaces to use the FleXR ports without considering how the ports will be configured by a user.

The port manager design is shown in Figure 4. The manager has input and output port maps. These port maps have the mapping information of the port tags registered by a developer and the FleXR ports activated with the communication attributes by a user. A kernel function can get inputs and send outputs via the port manager interfaces with the tag. The branched port map contains the ports branched from the registered output port. When a registered output port needs to be connected into multiple downstream ports with different communication attributes, the port manager activates the branched ports and keeps their mapping information to the registered port. When a kernel function sends an output to the registered port, it is also sent through the branched ports by using this mapping information.

Listing 1 shows the codes of an example kernel which a developer implements (❶ in Figure 3). In Line 4-6, the developer registers the input and output ports with the tags. The registered ports are used in Line 10-16 without specifying their connection types and branching states. The port manager hides the complexities of using the dynamically instantiated ports from developers.

FleXR Port abstracts different local and remote communication ports and exposes a unified interface to the port manager. When the pipeline manager creates a pipeline (❹ and ❼ in Figure 3), a kernel is instantiated and its ports are configured by the port manager with user-specified port connectivity, semantics, and recency management mechanism. Since these communication attributes are determined by the contexts of the requested pipeline, the operation of a FleXR port should differ based on the attributes given at runtime. We design the FleXR port abstraction as a state machine with the integrated interfaces.

The design of a FleXR port is shown in Figure 4. Each FleXR port has the port semantics, connection state, and local and remote ports. The port semantics is for specifying the communication semantics: blocking and non-blocking. The connection state indicates whether it is local or remote, and the network protocol for the remote. The

```
1  class ExampleKernel: public FleXRKernel {
2    public:
3      ExampleKernel() {
4        portManager.registerInPortTag("in1", PortSemantics::BLOCKING);
5        portManager.registerInPortTag("in2", PortSemantics::NONBLOCKING);
6        portManager.registerOutPortTag("out");
7      }
8
9      raft::kstatus run() {
10       MsgType *in1=portManager.getInput<MsgType>("in1");
11       MsgType *in2=portManager.getInput<MsgType>("in2");
12       MsgType *out=portManager.getOutputPlaceholder<MsgType>("out");
13
14       /* Kernel Functionality ... */
15
16       portManager.sendOutput("out", out);
17     }
18 }
```

Listing 1: An example kernel with two input and one output ports registered by a developer.

local and remote ports are the actual communication channels internally used and interfaced by the FleXR port abstraction. Since the FleXR port is an abstraction for different communication ports, it is extensible. New network protocols and local channels can be seamlessly integrated into distributed XR pipelines.

Listing 2 is part of an example pipeline recipe which a user provides (❷ in Figure 3). The user creates a pipeline by specifying the kernels, their port attributes in Line 5, 7-8, 11-12, and 14-15 and connections in Line 17-22. When the pipeline is created, the port manager activates the FleXR port. The activation instantiates an underlying channel corresponding to the specified attributes, and the channel is interfaced via the FleXR port. The FleXR port provides uniform interfaces to the port manager while behaving differently based on the underlying channel.

```
1  - kernel :  ExampleKernel
2    id :      example_kernel1
3    input :
4      - port_name: in1
5        connection_type: local
6      - port_name: in2
7        connection_type: remote
8        remote_info: [RTP, 14802]
9    output :
10     - port_name: out
11       connection_type: local
12       semantics: blocking
13     - port_name: branched_out
14       connection_type: remote
15       remote_info :  [127.0.0.1, 14805, TCP]
16       branched_from: out
17 - local_connections:
18   - send_kernel: example_kernel1
19     send_port_name: out
20     recv_kernel: example_kernel2
21     recv_port_name: input
22     queue_size: 1
```

Listing 2: A part of the pipeline recipe for the example kernel in Listing 1 and a connection.

4.3 Communication Semantics and Data Recency Management

To express the relationships among kernels and their dependencies and synchronization requirements, both blocking and non-blocking semantics are necessary for the local and remote communication primitives. FleXR supports the required semantics. The local communication in FleXR is based on the RaftLib port. Since the send and receive primitives of the vanilla RaftLib port are only with blocking semantics, we extend them with non-blocking semantics by checking the queue buffer of the connected RaftLib ports. A non-blocking send does not wait and continues when the queue connected to the downstream kernel is full. A non-blocking receive continues without waiting when the queue to the upstream kernel is empty. For remote communications, the socket and protocol implementations have interfaces with different semantics, and we map the underlying port interfaces to the FleXR port.

The data recency management mechanism in FleXR is to prevent data from aging in the pipeline queues. For local, FleXR provides recency management by limiting the number of messages in the queue buffer, which puts a bound on the maximum queuing delay [35]. The recency management for remote communication is done by supporting different network protocols, currently supporting TCP and RTP over UDP. For TCP connection, the in-order and reliable delivery may lead to lower data timeliness due to its retransmission and acknowledgment mechanisms. RTP over UDP has the advantage for data recency at the cost of data loss. By supporting these different protocols and queue size management, the recency management is achieved for remote and local communications.

4.4 Register-Activation Interface and Port-level Configuration

We embody the necessary DSP features for XR in the FleXR kernel, but these features should be provided properly to the stakeholders for supporting the runtime flexibility in distributed XR pipelines. The kernel developers know the input dependencies of their kernel functions, but it is unknown to them how their kernels are used in a pipeline which a user creates. When requesting a pipeline, the user arranges the pipeline structure with the kernel communication attributes. So, the user determines how the kernels operate within the pipeline. To provide the features to the proper stakeholder, FleXR has register-activation interfaces of the port manager at a port granularity, which streamline the development and deployment phases but clearly separate the features provided to each phase.

Based on the information available to the development and deployment phases, we identify the proper stakeholder for each feature and make the interfaces expose it. Table 3 summarizes the provided interfaces to each stakeholder. The developers register ports and set the input-port dependencies as they know the kernel functionalities. The connection type, branching outstream, output semantics, and recency management are specified by the user recipe because these attributes should be configured when the port manager activates FleXR ports by the metadata of the user pipeline.

The register-activation interfaces are enabled by the FleXR port and port manager, and the stakeholders use the FleXR features through them. When the developer registers the ports, the semantics of input ports are set via the port manager as shown in Line 4-5 of Listing 1. The connection types, recency management, and output semantics are specified by the user recipe as shown in Listing 2. The user can branch a single registered port with separate attributes in Line 13-16 of Listing 2. When the pipeline manager instantiates

Table 3: FleXR interface availability for the stakeholders to manipulate the features to resolve the DSP issues in §3.

Feature \ Stakeholder	Developer	User
Port registration	✓	
Port activation		✓
Output branching		✓
Input semantics	✓	
Output semantics		✓
Recency management		✓

the pipeline kernels, the user-specified attributes and branching are set for each port by the port manager.

5 IMPLEMENTATION

The current version of FleXR is implemented and tested on Ubuntu 20.04. It is written in C++ with STL, and on-node kernel management and communication rely on RaftLib v0.7 [4]. For remote communication, FleXR supports TCP using ZeroMQ [25], and RTP communication using uvgRTP [2] which is based on the RFC 3550 specifications [53]. For the application functionalities, we use several components. For object detectors in the AR applications, we use the ArUco [50] and ORB keypoint detection algorithm of OpenCV4 [58]. Pose estimation in the VR application is implemented using ORB SLAM3 [8] and the EuRoC dataset [7]. FleXR supports the Unreal Engine 5 (UE5) [18] and Unity 3D [59] game engines, which interface with the FleXR runtime via plugins. The FleXR plugins are compatible with the shared memory port in Figure 4 and make it possible to run external applications with FleXR pipelines. The graphics rendering in our examples uses the Mesa implementation [29] of OpenGL [28], EGL [26], and Vulkan [27]. For hardware-accelerated encoding and decoding of H.264 [62], we use FFmpeg [15], NVIDIA Video Codec [45] on the server, and NVIDIA L4T [46] on Jetson.

6 EVALUATION

The main objective of FleXR is to bring flexibility in distributed XR for realizing effective server assistance to XR use cases. For evaluation, we implement three XR use cases in Figure 5 and set four distribution scenarios in Figures 6 and 7. We compare FleXR to the existing distributed XR platforms in the ability to support the distribution scenarios. For each case, we evaluate the offloading impact in terms of pipeline latency and throughput. Additionally, we evaluate the benefit of the design of FleXR compared to existing thread-level SP frameworks: GStreamer [21] and RaftLib [4].

6.1 Experimental Testbed

In our setup, the client is NVIDIA Jetson AGX Xavier [43] with 8 core ARMv8 CPU, Volta GPU, and 32 GB memory shared by CPU and GPU. The Jetson runs in 15W and 30W power modes (Jet15W using 4 cores and Jet30W using 8 cores). The server has Intel Core i7-10700, 32 GB memory, and NVIDIA RTX 2070 of 8 GB GDDR6 memory. The server and client are connected via Gigabit Ethernet of 1 Gbps bandwidth with round-trip time (RTT) of 1.5 ms.

6.2 XR Applications and Distribution Scenarios

Example XR Applications. We evaluate the effectiveness of FleXR with 2 AR and 1 VR applications as shown in Figure 5. All the applications generate rendered frames of 1080p and provide Full HD (FHD) experiences. The AR use cases have the same pipeline

Table 4: The supportability of the existing frameworks and FleXR to our distribution scenarios in Figure 6 and 7.

	`Local`	`Perceptions`	`Rendering+App`	`Full Offloading`
Marvel [9]	✗	✓	✗	✗
OpenRiST [20]	✗	✗	✗	✓
Furion [33]	✗	✗	✓	✗
Liu *et al.* [36]	✗	✓	✗	✗
Liu *et al.* [37]	✗	✗	✓	✗
Schneider *et al.* [52]	✗	✗	✗	✓
Zhang *et al.* [66]	✗	✗	✓	✗
FleXR	✓	✓	✓	✓

structure in Figure 6a taking 1080p camera frames, but with different workload characteristics. The camera frames are branched to the object detector and renderer because the camera frame needs to be rendered as a background. The renderer receives the background frame in a blocking manner. The connection for the object pose detector is non-blocking as an object might not be detected.

For the first AR case (AR1) in Figure 5a, the object detection is done by the local feature matching processes: ORB feature extraction [51], k-nearest neighbor (KNN) descriptor matcher, homography and transformation estimations via a perspective-n-point (PnP) random sample consensus (RANSAC) solver [16]. The second AR case (AR2) in Figure 5b uses the ArUco algorithm [19] for detecting the fiducial markers. While AR2 has a less complex perception than AR1, the application and rendering are more intensive in AR2 as the application is implemented as a separate process by UE5 of the physics and shaders. On the other hand, AR1 rendering is a pipeline kernel and uses only low-level 3D graphics APIs.

For the VR use case in Figure 5c, the pose estimator gets 480p camera frames and inertial measurement unit (IMU) data and generates the current user pose as shown in Figure 7a. The pose estimator of the monocular-inertial SLAM has the primary input of IMU and the camera input is optional. The renderer shows the 3D scene captured from the estimated user pose. For all example use cases, the user can interact with the virtual objects via keyboard inputs, and this interaction is done by non-blocking receives because the key event happens arbitrarily by the user.

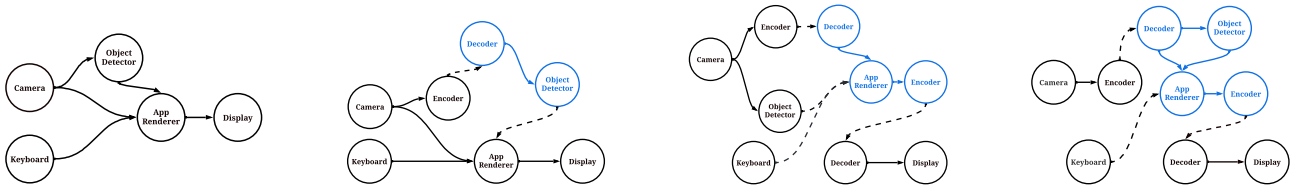
Distribution Scenarios. We set up four distribution scenarios for the three use cases: `Local` (**L**), `Perception` (**P**), `Rendering+App` (**R**), and `Full Offloading` (**P+R**). The canonical XR applications consist of perception and graphics rendering functionalities. As summarized in Table 1, the existing distributed XR systems can be categorized into one of our scenarios by their offloading supportability. With our scenarios, we show the flexibility benefit of FleXR compared to the existing systems.

In `Local` (**L**), all functionalities run local on the client device only. In `Perception` (**P**), only the perception kernels are offloaded to the server, and in `Rendering+App` (**R**) the application rendering are offloaded. In `Full Offloading` (**P+R**), the client only sends the sensor data and receives the final rendered frame. Figures 6 and 7 show the configurations for the AR1/2 and VR scenarios. Compared to existing work which targets specific distributed XR configurations, FleXR can enable all distribution scenarios flexibly, as shown in Table 4.

For the distributed configurations of AR1/2 and VR, the camera and rendered frames and IMUs are transferred with RTP over UDP for application responsiveness while the user input from the keyboard is with TCP for reliable delivery. When the sensor data is

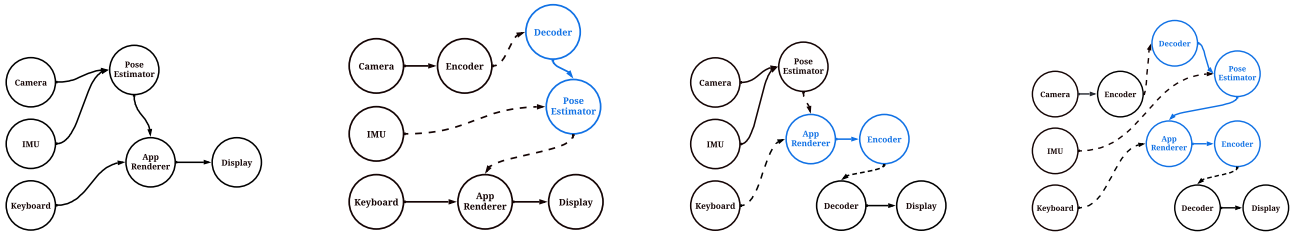
(a) The first AR use case (AR1). **(b)** The second AR use case (AR2). **(c)** The VR use case (VR).

Figure 5: The screenshots of the example use cases.



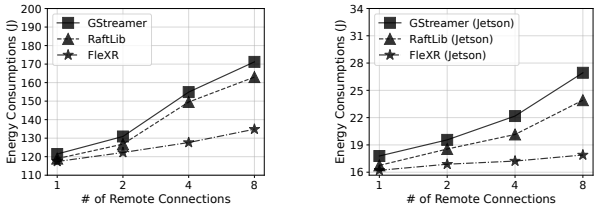
(a) `Local`: the pipeline runs locally. **(b)** `Perception`: the perception kernels run on the server. **(c)** `Rendering+App`: the application and rendering run on the server. **(d)** `Full Offloading`: all functionalities run on the server.

Figure 6: The distribution scenarios of AR use cases (blue parts on the server).



(a) `Local`: the pipeline runs locally. **(b)** `Perception`: the perception kernels run on the server. **(c)** `Rendering+App`: the application and rendering run on the server. **(d)** `Full Offloading`: all functionalities run on the server.

Figure 7: The distribution scenarios of VR use case (blue parts on the server).



(a) Energy usage results on the server machine. **(b)** Energy usage results on Jetson 30W.

Figure 8: Energy consumption to send 1000 messages of 512 Bytes every 10 ms to remote kernels on our server and Jetson 30W.

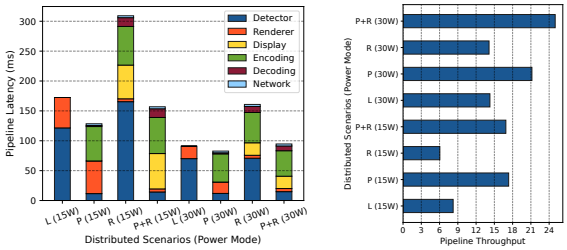
moved via local connections, its queue size is set as 1 to minimize the queuing delay for the data recency.

6.3 Design Benefit of FleXR

As shown in Table 2, GStreamer and RaftLib provide efficient local communication. Instead of using FleXR, a distributed pipeline can be supported by implementing auxiliary kernels for remote communications, branching, and synchronization. However, supporting a distributed pipeline with the auxiliary kernels introduces inefficiencies because each kernel, including the auxiliary ones, is a separate execution unit that is parallelly scheduled and managed.

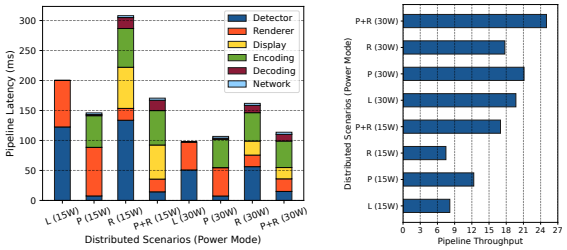
Figure 8 shows the overheads of the auxiliary kernels on our testbed. We create a kernel sending output to multiple remote kernels, and measure the energy consumption for the transmissions. GStreamer and RaftLib need the additional kernels for remote messaging and output branching; for sending output to 8 remote kernels, 9 auxiliary kernels are required (1 for branching and 8 for remote messaging). The energy consumption with the auxiliary kernels increases with their number. In contrast, with the FleXR kernel design and interface, the output port registered by a developer can be branched and configured for remote connections with different protocols as specified by a user, not requiring additional kernels. In FleXR, the developers also don't need to implement these auxiliary kernels to make their kernels operate flexibly.

Table 5 shows the number of kernels for GStreamer, RaftLib, and FleXR to support AR1 in the different scenarios. GStreamer and RaftLib need auxiliary kernels as local SP libraries. GStreamer, as a multimedia framework, imposes strict synchronization across



(a) The latency breakdowns of the pipeline components. **(b)** The pipeline throughputs.

Figure 9: The pipeline latencies and throughputs of AR1 in our distribution scenarios.



(a) The latency breakdowns of the pipeline components. **(b)** The pipeline throughputs.

Figure 10: The pipeline latencies and throughputs of AR2 in our distribution scenarios.

streams based on their internal timestamps and requires all kernels to have a single synchronized stream for input and output [57]. Since RaftLib does not require such strict synchronization, it requires fewer kernels. GStreamer requires two kernels for branching the camera stream and synchronizing streams for the renderer, while RaftLib only needs a branching kernel. For the distributed settings, both need additional messaging kernels: 4 for `Perceptions`, 8 for `Rendering+App`, and 6 for `Full Offloading`. In FleXR, these auxiliary kernels are not required because each port can be configured for different usage, which enables the various scenarios flexibly without the system overheads from the auxiliary kernels.

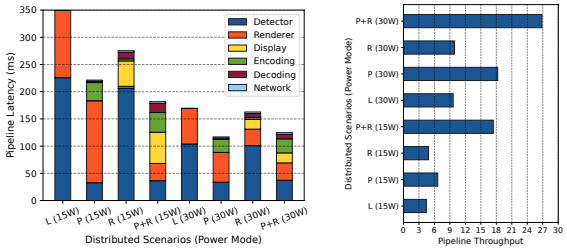
6.4 Evaluation of Example Applications

We run the example applications on our testbed of Jetson 15W and 30W with the four distribution scenarios to emulate the situations where the client has little or moderate device capacity. We measure the average pipeline latency and throughput of the three examples with the scenarios, and demonstrate that the flexibility enabled by FleXR makes it possible to achieve effective server offloading.

Costs for XR Pipeline Distribution. Distributing XR pipelines incur additional costs: multimedia data compression [22], displaying the rendered scene from a server, and network transmission [30]. Transmitting the large multimedia data without compression causes

Table 5: The number of kernels required to support the distributed configurations of AR1 in Figure 6.

	`Local`	`Perceptions`	`Rendering+App`	`Full Offloading`
GStreamer [21]	7	13	19	17
RaftLib [4]	6	12	18	16
FleXR	5	7	9	9



(a) The latency breakdowns of the pipeline components. **(b)** The pipeline throughputs.

Figure 11: The pipeline latencies and throughputs of VR in our distribution scenarios.

high bandwidth usage with backend network delays and consumes the battery of the user device [61, 63]. Therefore, data (de)compression is necessary for both the client and server. Another cost is for the client to display the rendered scene from the server. When the scene is rendered on the server, it should be fetched from GPU memory, sent to the client, and displayed.

The results in Figures 9-11, show a breakdown of the average end-to-end latency and average throughput. The display latency on the client is shown separately from the rendering latency. The compression cost is split as encoding and decoding latencies, which include the server- and client-side compression latencies. The network transmission latency is measured on the client when it receives the result of the timestamped message from the server.

Pipeline Latency and Throughput. Figures 9-11 show the latency and throughput results of AR1/2 and VR in the distribution scenarios (**L**, **P**, **R**, and **P+R**). Latency is measured as how long the pipeline takes to reflect the real-world context, and throughput is how frequently the real-world context is reflected to the rendered scene per second.

For AR1 (Figure 9), **P** shows the lowest latencies in 15W and 30W. For throughputs, **P** has the highest throughput in 15W while **P+R** does in 30W. Since the pipeline throughput is bound by the dominant functionality, it is possible to have lower throughput even with lower latency. Compared to **L**, the throughputs can be improved 2.1× (15W) and 1.7× (30W), and the latencies reduced by 28% (15W) and 14% (30W).

For **P**, the perception on the server takes 11 ms; it takes 121 ms (15W) and 70 ms (30W) of **L** on the client. Rendering on the client takes 54 ms (15W) and 19 ms (30W). **P** requires the client to encode 1080p camera frames, which takes 57 ms (15W) and 47 ms (30W). Decoding on the server takes 1.8 ms. The throughputs of **P** in 15W and 30W are bound by the encoding latency.

For **P+R**, since all rendering and perception run on the server, requires client-side displaying, encoding, and decoding. On Jet15W, it takes 59 ms for displaying the received FHD scene from the server, 57 ms for encoding camera frames, 12 ms for decoding the received scene. On Jet30W, it takes 20 ms for displaying, 40 ms for encoding, 6 ms for decoding. On the server, it takes 14 ms for perception, 5 ms for rendering, 1.7 ms for decoding the received camera frame, and 5 ms for encoding the rendered scene. So, the throughput of **P+R** (15W) is bound by client-side displaying while the throughput of **P+R** (30W) is bound by client encoding.

For AR2 (Figure 10), **P+R** shows the highest throughputs while **P** (15W) and **L** (30W) present the lowest latencies. The throughput of **P+R** is 1.5× of **P** (15W) and 1.3× of **L** (30W), but it has increase in latencies of 15% (15W) and 8% (30W).

For **P** (15W), the perception takes 7 ms on the server while it does 122 ms of **L** (15W), and the rendering takes 81 ms. In addition, it takes 52 ms for client encoding and 1.8 ms for server decoding. For **L** (30W), it takes 51 ms for perception and 47 ms for rendering.

For **P+R**, on Jet15W, it takes 54 ms for encoding the camera, 13 ms for decoding the rendered scene, and 57 ms displaying the received scenes while taking 14 ms for perception and 20 ms rendering on the server. On Jet30W, it takes 40 ms for encoding, 7 ms for decoding, and 18 ms displaying. While the throughputs of **P** (15W) and **L** (30W) are bound by rendering and perception each, the throughputs of **P+R** in 15W and 30W are bound by the client encoding.

In Figure 11 of VR, **P+R** (15W) and **P** (30W) present the lowest latencies. **P+R** shows the highest throughputs in 15W and 30W. Compared to **L**, the throughputs can be 3.9× (15W) and 2.7× (30W), and the latencies are 50% less (15W) and 29% less (30W).

For **P**, on Jet30W, it takes 54 ms for rendering, 24 ms for encoding 480p camera frames. On the server, it takes 33 ms for perception and 1.5 ms for decoding the received camera frames. On Jet15W, it takes 150 ms for rendering and 33 ms for encoding. The throughputs of **P** in 15W and 30W are bound by the rendering. Since the rendering is so challenging for Jet15W, it dominates throughput and latency.

For **P+R**, on Jet15W, it takes 57 ms for displaying the scene from the server, 31 ms for encoding camera frames, 15 ms for decoding the received scene. On Jet30W, it takes 18 ms for displaying, 20 ms for encoding, 7 ms for decoding. On the server, it takes 36 ms for perception, 31 ms for rendering, 1.7 ms for decoding the received camera frame, and 5 ms for encoding the rendered scene.

For Jet15W, rendering and perception are challenging, and **P+R** is beneficial in terms of latency and throughput. In the case of Jet30W, even though **P+R** introduces additional overheads for the client-side decoding and displaying, the pipeline bottleneck of rendering is relieved compared to **P** (30W), enabling higher throughput.

Result Analysis. In the results in Figure 9, the optimal distribution scenario can vary in terms of the latency and throughput even with the same workloads by the client capacity: **P** and **P+R** in 15W and 30W. For higher throughput, it is crucial to offload the pipeline bottleneck. There are cases where the distribution overheads are larger than the benefits, ending up with worse performance than the local-only scenario (*e.g.*, **R** (15W) and (30W) of AR1 and AR2 in Figure 9 and 10). The server and client device capacities should be considered because the kernels are parallelized and can cause resource contention. For instance, the perception latencies of **P+R** in AR1, AR2, and VR increase on the server compared to **P**. Moreover, although AR1 and AR2 are with the same pipeline structure of Figure 6, the ideal distribution is different based on the perception and rendering complexities of each application.

Based on our results, the effectiveness of offloading depends on the given workloads, server and client capacities, and offloading overheads. FleXR allows each user to configure the workload distribution flexibly at runtime and enables the optimal distribution of an XR application for various distribution scenarios.

7 DISCUSSION

FleXR offers flexibility in how an XR workload is distributed across a device and offload server(s). This opens up several opportunities for innovation and new research directions.

First, while with FleXR the XR configuration can be tuned to the specifics of the deployment context to realize optimal workload distributions, currently, this requires manual effort from the system users. To fully realize the potential of FleXR, future research is needed on automated deployment and resource management. New methods are needed to consider factors such as kernel costs, client and server capacity, network state, and offloading overheads, as well as to enable dynamic adaptation of the workload configurations. Previous function offloading systems have used linear solvers and resource profilers for offloading decisions with static analysis [11, 13], but this approach would be more complex in FleXR as the kernel costs and offloading overheads are subject to change based on user and server situations.

Second, by making it possible to integrate third-party components and application frameworks with FleXR (*e.g.*, game engines), we make it possible to consider a future landscape of XR supported by distributed edge-cloud infrastructure, potentially with different performance, quality, or other properties, that can be combined in different ways to support complex future XR use cases. This new landscape opens up new challenges for distributed orchestration for XR, and also promotes the reuse of service functionality in different scenarios, thus enabling faster innovation.

Finally, while distributed service composition has been considered in other contexts [5, 24, 64], several XR-specific aspects raise new challenges and opportunities that future work should address. These are related to performance/timeliness and quality tradeoffs that exist at the application level, sharing and reuse across users (*e.g.*, as done for specific offload services in [48]), XR-specific transport protocols [1, 6], new privacy concerns, *etc.* By creating and open sourcing the FleXR infrastructure, we believe our work will facilitate such research directions.

8 CONCLUSION

In this work, we describe the limitations of the existing distributed XR systems. We argue the need for flexibility in XR workload distribution, and that the lack of flexibility in distributed XR is attributed to the absence of adequate system supports. To address this, we present FleXR – a DSP system enabling the flexible distribution of XR pipelines at runtime. We identify several issues with applying existing DSP designs to XR, and resolve them while building FleXR. Our experimental evaluation demonstrates that FleXR efficiently enables different distribution scenarios for three XR use cases. The results show the optimal workload distribution is determined by environmental factors, and FleXR makes it possible each XR use case to benefit from edge server assistance in different environments.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers. This work has been partially supported by NSF projects CCF-2217070 and CNS-1909769, the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, and by funding and equipment gifts from VMware and Intel.

REFERENCES

[1] Maha Abdallah, Carsten Griwodz, Kuan-Ta Chen, Gwendal Simon, Pin-Chun Wang, and Cheng-Hsin Hsu. 2018. Delay-sensitive video computing in the cloud: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 14, 3s (2018), 1–29.

[2] Aaro Altonen, Joni Räsänen, Jaakko Laitinen, Marko Viitanen, and Jarno Vanne. 2020. Open-Source RTP Library for High-Speed 4K HEVC Video Streaming. In *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. IEEE, 1–6.

[3] Pablo Basanta-Val, Norberto Fernandez-Garcia, Luis Sanchez-Fernandez, and Jesus Arias-Fisteus. 2017. Patterns for distributed real-time stream processing. *IEEE Transactions on Parallel and Distributed Systems* 28, 11 (2017), 3243–3257.

[4] Jonathan C Beard, Peng Li, and Roger D Chamberlain. 2017. RaftLib: a C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications* 31, 5 (2017), 391–404.

[5] Ketan Bhardwaj, Sreenidhy Sreepathy, Ada Gavrilovska, and Karsten Schwan. 2014. ECC: Edge Cloud Composites. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. 38–47. https://doi.org/10.1109/MobileCloud.2014.18

[6] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. 2017. Future networking challenges: The case of mobile augmented reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1796–1807.

[7] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. 2016. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research* 35, 10 (2016), 1157–1163.

[8] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. 2021. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics* 37, 6 (2021), 1874–1890.

[9] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E Culler, and Randy H Katz. 2018. Marvel: Enabling mobile augmented reality with low energy and low latency. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 292–304.

[10] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 155–168.

[11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. 301–314.

[12] Cristian Garcia. 2018. Pypeln, A simple yet powerful Python library for creating concurrent data pipelines. https://cgarciae.github.io/pypeln/.

[13] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. 49–62.

[14] Robert Cypher and Eric Leu. 1994. The semantics of blocking and nonblocking send and receive primitives. In *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 729–735.

[15] FFmpeg team. 2021. FFmpeg, A complete, cross-platform solution to record, convert and stream audio and video. https://www.ffmpeg.org/.

[16] Martin A Fischler and Robert C Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* 24, 6 (1981), 381–395.

[17] Python Software Foundation. 2021. multiprocessing.shared memory, Provides shared memory for direct access across processes. https://docs.python.org/3/library/multiprocessing.sharedmemory.html.

[18] Epic Games. 2022. Unreal Engine: The most powerful real-time 3D creation platform. https://www.unrealengine.com.

[19] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. 2014. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition* 47, 6 (2014), 2280–2292.

[20] Shilpa George, Thomas Eiszler, Roger Iyengar, Haithem Turki, Ziqiang Feng, Junjue Wang, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2020. OpenRTiST: End-to-End Benchmarking for Edge Computing. *IEEE Pervasive Computing* 19, 4 (2020), 10–18.

[21] GStreamer Team. 2001. GStreamer: a flexible, fast and multiplatform multimedia framework. https://gstreamer.freedesktop.org/.

[22] Jin Heo, Christopher Phillips, and Ada Gavrilovska. 2022. FLiCR: A fast and lightweight lidar point cloud compression based on lossy ri. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE, 54–67.

[23] Holo-Light. 2020. ISAR SDK – XR Streaming. https://holo-light.com/products/isar-sdk/.

[24] Songlin Hu, Vinod Muthusamy, Guoli Li, and Hans-Arno Jacobsen. 2008. Distributed Automatic Service Composition in Large-Scale Systems. In *Proceedings of the Second International Conference on Distributed Event-Based Systems* (Rome, Italy) *(DEBS '08)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1385989.1386019

[25] iMatix. 2021. ZeroMQ, An open-source universal messaging library. https://zeromq.org/.

[26] The Khronos Group Inc. 2014. EGL, Native Platform Interface. https://www.khronos.org/egl/.

[27] The Khronos Group Inc. 2016. Vulkan, Cross platform 3D Graphics. https://www.vulkan.org/.

[28] The Khronos Group Inc. 2017. OpenGL, The Industry's Foundation for High Performance Graphics. https://www.opengl.org/.

[29] VMware Intel, AMD. 2021. The Mesa 3D Graphics Library. https://www.mesa3d.org/.

[30] Minsung Jang, Karsten Schwan, Ketan Bhardwaj, Ada Gavrilovska, and Adhyas Avasthi. 2014. Personal Clouds: Sharing and Integrating Networked Resources to Enhance End User Experiences. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2220–2228. https://doi.org/10.1109/INFOCOM.2014.6848165

[31] Shweta Khare, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Aniruddha Gokhale, Yogesh Barve, Anirban Bhattacharjee, and Xenofon Koutsoukos. 2019. Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 1–14.

[32] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE Infocom*. IEEE, 945–953.

[33] Zeqi Lai, Y Charlie Hu, Yong Cui, Linhui Sun, Ningwei Dai, and Hung-Sheng Lee. 2019. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. *IEEE Transactions on Mobile Computing* 19, 7 (2019), 1586–1602.

[34] Mengtian Li, Yu-Xiong Wang, and Deva Ramanan. 2020. Towards streaming perception. In *European Conference on Computer Vision*. Springer, 473–488.

[35] John DC Little. 1961. A proof for the queuing formula: L= λ W. *Operations research* 9, 3 (1961), 383–387.

[36] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[37] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. 2018. Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 68–80.

[38] Xing Liu, Christina Vlachou, Feng Qian, Chendong Wang, and Kyu-Han Kim. 2020. Firefly: Untethered Multi-user VR for Commodity Mobile Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 943–957. https://www.usenix.org/conference/atc20/presentation/liu-xing

[39] Jiayi Meng, Sibendu Paul, and Y Charlie Hu. 2020. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 923–937.

[40] Microsoft. 2019. Azure Custom Vision. https://azure.microsoft.com/en-us/services/cognitive-services/custom-vision-service.

[41] Microsoft. 2020. Azure Remote Rendering. https://azure.microsoft.com/en-us/services/remote-rendering/.

[42] Diego González Morín, Pablo Pérez, and Ana García Armada. 2022. Toward the Distributed Implementation of Immersive Augmented Reality Architectures on 5G Networks. *IEEE Communications Magazine* 60, 2 (2022), 46–52.

[43] Nvidia Corporation. 2018. Jetson AGX Xavier Developer Kit. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[44] Nvidia Corporation. 2020. NVIDIA CloudXR™ SDK. https://developer.nvidia.com/nvidia-cloudxr-sdk.

[45] Nvidia Corporation. 2021. NVIDIA Video Codec SDK. https://developer.nvidia.com/nvidia-video-codec-sdk.

[46] Nvidia Corporation. 2022. NVIDIA Jetson Linux Developer Guide : Introduction. https://docs.nvidia.com/jetson/l4t.

[47] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[48] Xukan Ran, Carter Slocum, Maria Gorlatova, and Jiasi Chen. 2019. ShareAR: Communication-efficient multi-user mobile augmented reality. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 109–116.

[49] Henriette Röger, Sukanya Bhowmik, and Kurt Rothermel. 2019. Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures. In *Proceedings of the 20th International Middleware Conference*. 255–267.

[50] Francisco J Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. 2018. Speeded up detection of squared fiducial markers. *Image and vision*

Computing 76 (2018), 38–47.
[51] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.
[52] Michael Schneider, Jason Rambach, and Didier Stricker. 2017. Augmented reality based on edge computing using the example of remote live support. In *2017 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 1277–1282.
[53] Henning Schulzrinne, Steven Casner, R Frederick, and Van Jacobson. 2003. RFC3550: RTP: A transport protocol for real-time applications.
[54] Henning Schulzrinne, Anup Rao, and Robert Lanphier. 1998. Real time streaming protocol (RTSP). (1998).
[55] Wright Stevens et al. 1997. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. (1997).
[56] Tarik Taleb, Zinelaabidine Nadir, Hannu Flinck, and JaeSeung Song. 2021. Extremely interactive and low-latency services in 5G and beyond mobile systems. *IEEE Communications Standards Magazine* 5, 2 (2021), 114–119.
[57] Wim Taymans, Steve Baker, Andy Wingo, Rondald S Bultje, and Stefan Kost. 2013. Gstreamer application development manual (1.2. 3). *Publicado en la Web* (2013).
[58] OpenCV team. 2021. OpenCV 4.0. https://opencv.org/opencv-4-0/.
[59] Unity Technologies. 2021. Unity Real-Time Development Platform. https://unity.com.
[60] Velimir Mlaker. 2018. MPipe, Multiprocess Pipeline Toolkit for Python. http://vmlaker.github.io/mpipe/index.html.
[61] Ekhiotz Jon Vergara and Simin Nadjm-Tehrani. 2013. EnergyBox: a trace-driven tool for data transmission energy consumption studies. In *European Conference on Energy Efficiency in Large Scale Distributed Systems*. Springer, 19–34.
[62] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. 2003. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology* 13, 7 (2003), 560–576.
[63] Yu Xiao, Yong Cui, Petri Savolainen, Matti Siekkinen, An Wang, Liu Yang, Antti Ylä-Jääski, and Sasu Tarkoma. 2013. Modeling energy consumption of data transmission over Wi-Fi. *IEEE Transactions on Mobile Computing* 13, 8 (2013), 1760–1773.
[64] Xiaofei Xu, Xiao Wang, Hanchuan Xu, and Zhongjie Wang. 2021. Distributed Service Composition in Internet of Services. In *2021 IEEE International Conference on Services Computing (SCC)*. 274–284. https://doi.org/10.1109/SCC53864.2021.00040
[65] Yu-Ping Wang. 2017. shm transport, The shared memory transport package. http://wiki.ros.org/shm_transport.
[66] Lei Zhang, Andy Sun, Ryan Shea, Jiangchuan Liu, and Miao Zhang. 2019. Rendering multi-party mobile augmented reality from edge. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 67–72.

A ARTIFACT APPENDIX

A.1 Abstract

This appendix presents the software artifact of the FleXR implementation that accompanies this paper. With this artifact, the concept and effectiveness of FleXR can be demonstrated – a user can create distributed XR pipelines with given kernels and different kernel connections, and it does not require any modification of the kernel codes. The latest version of the artifact is available at https://github.com/gt-flexr/FleXR, and the corresponding documentation is at the wiki page of this repository.

A.2 Artifact check-list (meta-information)

- **Program: FleXR**
- **Compilation: CMake (> 3.10), Makefile (> 4), and gcc/g++ (> 9)**
- **Data set: Camera frames with ArUco markers**
- **Run-time environment: Ubuntu 18.04/20.04, X11, OpenGL, OpenCV4, RaftLib, Docker**
- **Hardware: CPU (x86 of Intel/AMD or ARM architecture), Nvidia GPU (optional)**
- **Execution: Running a distributed pipeline that is described by a yaml file**
- **Metrics: Pipeline throughput and latency**
- **Output: The display of rendered frames of an AR application and the logged latency and throughput**
- **Experiments: Demonstration of running an AR example with 4 different distributed configurations on a local machine**
- **How much disk space required (approximately)?: ~6 GB with our CPU-only docker image, ~12 GB with our GPU-enabled docker image**
- **How much time is needed to prepare workflow (approximately)?: < 30 mins with our docker images**
- **How much time is needed to complete experiments (approximately)?: < 1 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT license**
- **Archived (provide DOI)?: 10.5281/zenodo.7824232**

A.3 Description

A.3.1 How delivered. We provide a set of docker images that contain all the dependencies of FleXR and the source code for the demonstration. While it is recommended to use the docker images, it is also possible to setup the environment manually (https://github.com/gt-flexr/FleXR/wiki/Installation-Instructions:-Ubuntu-18.04-&-20.04). The docker images are available via Docker Hub (https://hub.docker.com/repository/docker/jheo4/flexr/general). Among the available images, `armv8_dev` is the CPU-only image for the ARM architecture. `dev_2004` is the CPU-only image for the x86 architecture of Intel/AMD. `nv_dev_2004_114` is the image supporting NVIDIA GPU of with CUDA 11.4.

All the images are based on Ubuntu 20.04 and contain the sample camera frames with ArUco markers and source codes. After compiling FleXR, the demo with an AR application and four distribution scenarios described in §6.2 can be run. The step-by-step tutorial for this demo is available in our wiki page (https://github.com/gt-flexr/FleXR/wiki/Get-Started:-Proof-of-Concept)

A.3.2 Hardware dependencies. FleXR is compatible with x86-/ARM-based CPUs. Among the XR kernels that we provide, there are kernels, *e.g.*, keypoint detection and video codecs, utilizing the hardware acceleration of NVIDIA GPUs. To use those kernels, a GPU with CUDA support is required. However, even when the GPU is not available, FleXR can be used with the

CPU-only kernels; at that time, the compile setting should be modified to disable the GPU-using kernels in CMakeLists.txt.

A.3.3 Software dependencies. OpenCV4, RaftLib, ZeroMQ, yaml-cpp, spdlog, uvgRTP, Boost, Catch2, FFmpeg, OpenGL/Vulkan.

A.3.4 Data sets. We provide a set of sample camera frames with ArUco markers, which are available in the container images and used by the AR pipeline.

A.4 Installation

In this instruction, we assume the host machine with a display is Ubuntu 18.04 or 20.04 and the docker is installed already. We present the installation steps with our docker images. For manual installation, please refer to our wiki page (https://github.com/gt-flexr/FleXR/wiki/Installation-Instructions:-Ubuntu-18.04-&-20.04). After running the container, you are set to run the demo.

1. Pull the docker image from Docker Hub. `docker pull jheo4/flexr:dev_2004` (x86 CPU-only), `docker pull jheo4/flexr:nv_dev_2004_114` (x86 and GPU-enabled), or `docker pull jheo4/flexr:armv8_dev` (ARM CPU-only).
2. Increase UDP buffer size on host machine. `sudo sysctl -w net.core.rmem_max=26214400` and `sudo sysctl -w net.core.rmem_default=26214400`
3. Run the container. `docker run -it --net=host --ipc=host -e DISPLAY=$DISPLAY -v /tmp/.X11-unix/:/tmp/.X11-unix/ --privileged --name flexr_poc jheo4/flexr:dev_2004`
4. If the FleXR directory does not exist in the home directory, get FleXR source code. `git clone` https://github.com/gt-flexr/FleXR.git
5. Check out to the prepared branch. `cd FleXR && git fetch origin mmsys/artifact && git checkout mmsys/artifact`
6. Compile FleXR. `rm -r build && mkdir build && cd build && cmake .. && make -j$(nproc)`

A.5 Experiment workflow

This workflow should be after the above installation steps.

1. If the environment is not with our docker images, get the sample images from our drive. After getting the images manually, the directory path in the following yaml files should be modified accordingly. (https://drive.google.com/file/d/1ObORroLVVRulgkr0CBT9CzOdlUHNGnwO/view?usp=sharing)
2. Run the local-only configuration. `~/FleXR/bin$ ./runner -y ../examples/poc/simple_local.yaml`
3. Run the perception-offloaded configuration. `~/FleXR/bin$ ./runner -y ../examples/poc/simple_perception.yaml`
4. Run the renderer-offloaded configuration. `~/FleXR/bin$ ./runner -y ../examples/poc/simple_renderer.yaml`
5. Run the full-offloading configuration. `~/FleXR/bin$ ./runner -y ../examples/poc/simple_full.yaml`

A.6 Evaluation and expected result

After running the experiments, the execution results of the pipelined kernels are logged in `~/FleXR/bin/flexr_logs`. Under the log directory, there are directories of the process IDs. Each directory contains the log files of the corresponding pipeline kernels. In each kernel log file, the execution time of each kernel is logged. Moreover, the message indices are also recorded with the timestamps, and the pipeline throughput can be derived from these records.

In this tutorial, all distribution configurations run on the local machine, and the remote communications via ports are the inter process communications via port interfaces. Thus, the results are not hugely different from the local-only configuration to the other configurations. To get the evaluation results of this paper, the experimental testbed of the server and client devices and experiment customization are required. Then, the yaml files under the example/exp directory can be used to run the experiments.

A.7 Experiment customization

FleXR can be used as an end-to-end framework for distributed XR. By enabling the flexible XR pipeline distribution, FleXR can facilitate the further development and research around distributed XR.

As presented in this tutorial, the pipeline distribution can be differently configured by the yaml files. Not only for the given example pipelines, but also for the customized pipelines, the yaml files can be customized for different distribution scenarios with other kernels that we implement. There are example yaml files in the example directory of the FleXR repository, and they can be used as a reference for the customization by the kernel users.

Furthermore, new kernels can be added to FleXR for new XR functionalities (*e.g.*, perceptions and rendering). The kernel developer guide is also available on our wiki page (https://github.com/gt-flexr/FleXR/wiki/Basic-Kernel-Developer-Guide). By implementing an XR functionality as a FleXR kernel, it can be used in diverse distributed scenarios of the customized experiments without any modification.