计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(12): 3042−3059, 2025

本期推荐

支持端边云多运行时协同应用的网程系统

俞子舒　王一帆　曾　琛　张星洲　彭晓晖　徐志伟
（中国科学院计算技术研究所　北京　100190）
（中国科学院大学　北京　100049）
（yuzishu@ict.ac.cn）

Grip System for Multi-Runtime Support in Things-Edge-Cloud Collaborative Applications

Yu Zishu, Wang Yifan, Zeng Chen, Zhang Xingzhou, Peng Xiaohui, and Xu Zhiwei
（*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
（*University of Chinese Academy of Sciences*, *Beijing* 100049）

Abstract　Researchers have proposed and implemented many distributed runtime systems to help users build distributed applications. These distributed runtime systems are usually only good at processing certain types of loads in specific scenarios. However, in the things-edge-cloud scenario, the components of things-edge-cloud collaborative applications have heterogeneous quality requirements, runtime environments, and heterogeneous communication protocols, making it difficult to use one runtime to build high-performance and robust things-edge-cloud collaborative applications. Deploying application components independently to different runtimes will increase the difficulty of application management and lack of unified performance and fault-tolerance support. The Grip system is proposed to address the problem. The Grip system supports the unified access and utilization of multiple runtimes by introducing a virtual runtime adapter layer and a virtual runtime API layer. These two virtual layers specify the interfaces that need to be implemented when accessing a runtime. The Grip system supports the unified management of multi-runtime applications through Griplet and Grip abstractions. It utilizes ownership methods to provide mechanisms for supporting user-defined fault tolerance and scaling policy. Experiments show that in the things-edge-cloud environment, compared with using a single runtime such as Ray, Docker, and Kubernetes, the Grip system reduces the average end-to-end latency by 31% to 77%, the 90th percentile tail latency by 25% to 78%, and the 95th percentile tail latency by 22% to 78%.

Key words　multiple runtimes; things-edge-cloud; unified access; unified management; distributed system

摘　要　研究人员针对不同的负载类型提出并实现了大量的运行时系统，帮助用户构建单机或分布式应用.在端边云协同场景中，由于应用各组件在保质要求、运行时环境和通信协议方面存在异构性，因此难以通过单一运行时构建性能出色且鲁棒的端边云协同应用.将应用的各个组件独立部署到不同的运行时会增加应用管理的难度，并且缺乏对性能和容错方面的统一支持.为解决上述问题，实现了网程系统，支持多种运行时的统一接入和使用.网程系统通过网元和网程抽象支持多运行时应用的统一管理，并基于所有权方法提供自定义容错和缩扩容策略的支持机制.实验表明，在端边云环境下，相比于使用 Ray, Docker, Kubernetes 等单一运行时，网程系统降低了 31%~77% 的平均端到端延迟、26%~78% 的 90 百分位

收稿日期：2024-08-13；**修回日期**：2025-04-15
基金项目：国家自然科学基金项目（62072434, U23B2004）
This work was supported by the National Natural Science Foundation of China (62072434, U23B2004).
通信作者：彭晓晖（pengxiaohui@ict.ac.cn）

尾延迟、22%~78% 的 95 百分位尾延迟.

关键词 多运行时；端边云；统一接入；统一管理；分布式系统

中图法分类号 TP316.4

DOI: 10.7544/issn1000-1239.202440676 **CSTR:** 32373.14.issn1000-1239.202440676

面对日益增长的数据处理需求和复杂的应用场景，如何快速构建高效运行且鲁棒的分布式应用，成为了分布式系统研究的核心挑战. 针对不同场景，学术界和产业界设计实现了大量的分布式和单机运行时系统，如 Hadoop[1], Spark[2], Ray[3], Kubernetes (K8s)[4], Docker 等，以降低开发者构建和部署分布式应用的难度. 这些运行时系统通常优先支持某类负载，并采用特定的容错和缩扩容策略. 例如 Hadoop 专注于大数据批处理，难以支持实时负载. Spark 通过内存计算提升数据处理速度，但不适合资源受限的物端设备使用. Ray 是面向数据中心的人工智能分布式运行时，可自动将单机 Python 程序扩展到集群，但其设计不适合端边云协同环境. K8s 是面向大规模容器化应用的分布式运行时系统，通过了大规模的生产测试且拥有优秀的开源生态，但缩扩容延迟较高[4-5]. Docker 支持部署和管理单机容器化应用，但本身不支持容错和分布式. 当一个应用采用多种运行时来执行其不同组件时，我们称之为多运行时应用. 例如，在端边云的多运行时协同应用中，物端组件采用 Python 运行时，边缘组件采用 Docker 运行时，云端组件则采用 Ray 运行时.

在端边云场景中，应用通常需要在物端、边缘和云端设备上运行不同的组件. 由于这些设备的计算能力和网络环境各异，以及组件的隔离粒度和性能需求不同，使用单一运行时难以高效运行应用的不同组件. 因此，需要为不同组件选择合适的运行时，以构建性能出色且鲁棒的端边云应用. 运行时的选取需考虑 3 种异构性：1)任务保质需求异构. 云上通常执行实时性要求低、需要大量计算资源的任务，例如大数据处理、大模型推理等. 边缘和端设备上的任务通常有更高的实时性和较少的资源需求[6]. 2)运行时环境异构. 云端应用的运行时通常是分布式的，如 Spark, Ray, K8s 等，而端侧的运行时一般是单机的，如 Python 解释器、T-REST[7] 等. 边缘端则会采用单机运行时或轻量级分布式运行时，如 KubeEdge[8]. 此外，不同抽象层次的运行时提供不同的任务卸载能力[9]和隔离能力. 3)通信协议异构. 云端通常具有最好的网络条件，节点间通信带宽大，通常超过 10 Gbps 甚至数百 Gbps[10]，丢包率几乎为 0，延迟通常小于 1 ms. 因此，云上的运行时通常采用 gRPC, HTTP 等基于 TCP 的协议，包括 K8s, Ray, Knative[11], SOCK[12] 等. 而物端之间以及物端和边缘之间的网络则通常具有通信带宽低、丢包率高、延迟高的特点[13]. 例如，802.11ac 协议(Wi-Fi 5)的单路峰值带宽为 866.6 Mbps，在中等信号强度下丢包率超过 10%，存在干扰时的延迟可达 10 ms 以上[14-15]. 5G 蜂窝网络的增强型移动宽带(enhanced mobile broadband, eMBB)的平均下载速度约为 300 Mbps，而平均上传速度只有约 40 Mbps[16-17]，延迟为 4 ms[18]. 5G 中的超高可靠低延迟通信(ultra-reliable and low latency communications, URLLC)技术的理论延迟低至 1 ms，但主要面向低传输带宽场景. 同时，由于功耗、距离、延迟等需求，物端也出现诸多新型物理层通信协议[19]和应用层通信协议[20]，这些新的协议尚未被集成到云端的运行时.

此外，开发者会为应用的不同组件选择对应成熟且具有完整生态的运行时，以提升开发速度，降低出错概率. 例如，使用 Ray 运行时开发深度学习相关组件，使用 Docker 运行时社区已制作的服务镜像等.

因此，为端边云协同应用的各组件选择合适的运行时能够提升性能，并充分利用现有的生态以提升开发效率. 如图 1 所示，现有端边云应用通常采用多种运行时分别开发、独立管理的方式，或使用一种分布式运行时统一开发和管理的方式. 第 1 种方式无法在系统层提供统一的应用生命周期管理、性能优化和容错支持. 第 2 种方式则会加大利用现有生态的难度且容易降低应用的运行性能. 目前，尚缺乏一种

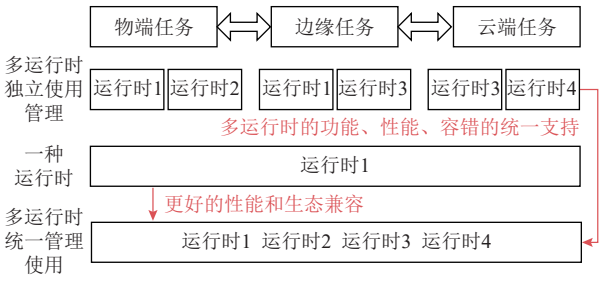


Fig. 1 Advantages of unified management of multiple runtimes

图 1 多种运行时统一管理的优势

支持单个应用使用多种运行时的统一管理系统. 文献 [21−22] 也指出智能万物互联时代的新型信息基础设施(算力网)应当具有类似于操作系统进程的分布式运行时抽象(网程), 使用不同的单机或分布式运行时来运行不同的模块, 并支持对各类算力网应用的全生命周期管理.

为了在系统层支持应用使用多种运行时, 本文设计实现了网程(Grip)系统. 该系统在不影响应用性能的同时, 为应用启动、缩扩容、错误恢复以及终止等生命周期管理提供系统级支持. 网程系统将应用各组件的执行实体统一抽象为网元(Griplet), 通过旁观式设计方法支持应用的高效执行, 并基于所有权(ownership)机制提供网元级自定义容错策略和缩扩容策略的支持.

网程系统的构建存在 4 个难点: 1)多种运行时的统一接入和使用. 每种运行时有不同的接口和使用方式. 例如, Python 解释器和 Docker 分别通过 exec 和 docker run 接口执行用户任务. 这将提高运行时统一接入和使用的难度. 2)性能保持与生态兼容. 采用网程系统构建和管理的应用, 其性能应与全人工构建和管理的应用性能一致, 且无需修改即可直接使用现有生态中的库和框架等资源. 3)统一生命周期管理. 可将分布式应用作为一个整体进行管理, 也可独立管理应用各组件. 4)灵活统一的容错和缩扩容支持机制. 网程系统应支持应用的不同组件采用自定义且各异的容错和缩扩容策略, 从而提升应用性能. 然而目前诸多端边云运行时并不提供缩扩容和容错的支持, 如 Python 解释器、Docker 等.

网程系统通过解耦端边云协同应用的执行与生命周期管理来解决上述难点. 图 2 展示了采用网程系统构建目标检测应用的示意图. 接下来简介网程针对上述难点的设计思路和方法.

1)多种运行时统一接入和使用. 网程系统引入虚拟运行时 API(以下简称虚运 API)与虚拟运行时适配器(以下简称虚运适配器)解决运行时的统一接入和使用问题. 虚运适配器从生命周期角度规定了运行时接入需要实现的接口集. 应用通过虚运 API 与虚运适配器交互, 从而使用不同的运行时. 如图 2 所示, 开发者基于虚运 API 与虚运适配器, 开发了 Python 运行时 API 与适配器, 从而将 Python 运行时接入至网程系统. 此外, 虚运适配器与 API 实现了应用元数据的上报、更新、查询和资源状态上报等非业务功能, 使得运行时 API 与适配器仅需关注业务逻辑, 以降低运行时接入的编程开销. 3.6 节提供了应用实例.

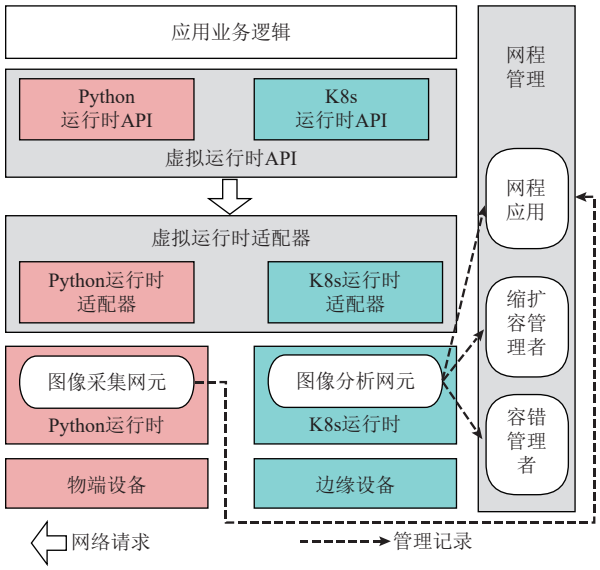


Fig. 2 Schematic diagram of object detection Grip application

图 2 目标检测 Grip 应用示意图

2)性能保持与生态兼容. 网程系统采用旁观式的设计方法, 既保持了应用的性能又兼容了现有的生态. 首先, 当计算任务提交到目标运行时后, 网程系统默认不介入任务的正常执行流程. 其次, 运行时适配器会直接在目标运行时上执行用户代码, 而不引入虚拟化、代理等中间层.

3)统一生命周期管理. 网程系统将任意单机或分布式运行时上执行的任务抽象为网元, 并规定了网元的控制块(control block)和生命周期管理接口(如终止、容错、健康检查等), 实现对应用组件的统一管理. 其次, 网元的控制块和网元间的创建关系构成了网程的控制块. 网程系统通过网程控制块实现应用的生命周期管理, 为用户提供网程的终止、缩扩容、查找等管理接口. 另外, 用户可基于网元接口实现自定义生命周期管理功能, 如终止网元及其关联的其他运行实例.

4)灵活的容错和缩扩容支持机制. 网程系统通过所有权机制为应用提供了网元级别的自定义容错和缩扩容策略的支持. 在网程系统中, 每个网元均可设置独立的容错管理者和缩扩容管理者以支持不同的容错和缩扩容策略. 当网元检测到其他网元出错时, 网程系统会自动向出错网元的容错管理者发出错误恢复请求. 系统采用基于签名的网元信息获取技术和递归式错误恢复机制, 支持网元及其容错管理者同时发生故障时, 仍可有效的恢复. 类似地, 当网元预估某个网元的服务延迟无法满足需求时, 网程系统会自动向网元缩扩容管理者发送扩容请求.

本文的贡献有 3 点:

1)提出并实现了面向端边云多运行时协同应用的网程系统. 系统通过网元和网程抽象为分布式应用同时使用多种运行时提供生命周期管理的支持.

2)引入了虚拟运行时 API 和虚拟运行时适配器, 支持多种运行时的统一接入和使用.

3)设计和实现了所有权机制, 为应用提供网元粒度的自定义容错和缩扩容策略支持.

实验结果显示, 基于网程系统实现的端边云协同应用, 与 Ray, Ray+K8s, Docker 等版本相比, 降低了 31%~77% 的平均端到端延迟、26%~78% 的 90 百分位尾延迟、22%~78% 的 95 百分位尾延迟. 网程应用的性能和全人工构建的应用性能相当, 实现了性能保持.

1 相关工作

为降低分布式编程难度、提升应用的性能和扩展性, 学术界和产业界提出和实现了多种分布式和单机运行时. Ray[3] 是面向人工智能领域的分布式编程框架, 可将单机 Python 程序自动扩展至集群. Ray 通过任务(task)和 Actor 编程抽象分别支持了无状态和有状态的分布式应用, 并通过 remote 操作将任务或 Actor 发送到远程节点执行. Ray 采用基于血统(lineage)的容错策略, 会取消所有使用该任务结果的其他任务, 并从某一个前置任务重新提交执行. 由于采用了 gRPC 通信协议, Ray 在网络环境不稳定时, 性能会大幅降低. K8s[4] 是目前广泛使用的云计算基础设施管理系统, 用于大规模容器化应用的部署和编排, 并提供自动缩扩容功能. K8s 允许用户在容器内使用各种单机运行时, 但其本身的开销较大, 不适合在边缘和物端设备上部署. Dapr[23] 是一款支持微服务和云原生应用的分布式运行时, 通过边车(side car)支持各类语言运行时的统一接入. 然而, 边车的代理转发会引入一定的开销. Dask[24] 是用于 Python 科学计算的分布式运行时, 可将单机的 Python 科学计算程序自动扩展到多机. Dask 在科学计算领域的性能优异, 但通用性不如 Ray 框架. SOCK[12] 和 Pagurus[25] 是为函数即服务(function as a service, FaaS)应用设计的分布式运行时, 大幅降低了冷启动开销, 适用于需要频繁缩扩容的场景.

KubeEdge[8] 针对边云协同计算, 将 K8s 的容器编排和管理能力扩展到边缘节点, 实现了云边消息的可靠传输及离线情况下的边缘自治. Azure IoT Edge[26] 是为边缘设备设计的运行时系统, 支持用户在边缘设备上部署容器化应用, 并通过云端接口控制边缘设备. IoT Edge 运行时提供了部署和更新计算任务、健康检查和边云通信等功能. 但它们只支持容器应用, 未提供多运行时支持.

工作流(workflow)系统允许不同阶段使用不同的运行时, 但它要求预先定义应用中各任务的执行顺序. 当前序任务完成后, 才可执行下一任务. 这一约束限制了应用的表达能力. 例如, 工作流系统不擅长表达终端采集数据、边缘预处理和云端分析的流式应用. 这类应用常表示为串行启动的工作流, 但不能使用该工作流进行后续的管理. 另外, 缩扩容操作也可能会破坏其依赖的有向无环图.

Apache Airflow[27] 是一个开源工作流管理平台, 它将工作流表示为任务的有向无环图. 调度程序遵循指定的依赖关系, 在多个工作节点上并行调度任务. Airflow 还支持自定义运算符和执行器, 以便于集成不同的运行时环境.

ExoFlow[28] 是一个通用的工作流分布式编程框架, 支持只执行 1 次的语义. 该框架允许用户自定义错误恢复函数和错误恢复类型, 实现了工作流执行与错误恢复的解耦. 其研究表明, 允许用户自定义错误恢复策略, 可显著提高应用的性能.

2 动机与问题

本节将从多运行时分布式应用的优势以及自定义缩扩容和容错策略的优势 2 个方面讨论研究动机, 并对研究问题进行形式化分析. 多运行时应用可根据应用组件的特征和需求, 选择适当的运行时, 降低软硬件环境异构性对应用的负面影响. 同时, 允许应用自定义各组件的缩扩容和容错策略, 可显著提升应用的性能与鲁棒性.

2.1 基于多运行时构建端边云应用的优势

在构建包含多类型任务的端边云应用时, 为任务选择合适的运行时可提高应用整体性能. 运行时通常是面向特定计算场景设计的, 它们在处理不同类型负载时的性能表现各异. 例如, Spark 可快速执行大数据负载; Ray 则专注于云端深度学习, 支持大规模训练和推理任务; K8s 是一款自动部署和管理容器化应用的分布式运行时; Docker 是一种轻量级的单机容器运行时, 适用于本地容器应用的部署和执行.

在某些场景下, 通过一种运行时构建包含多种异构负载的应用难以实现最佳性能. 为验证这一观点, 本文对比了单独使用 Ray 运行时、单独使用

Docker 运行时和混合使用 2 种运行时部署人脸识别应用的性能表现. 该人脸识别应用包含位于边缘的 VGG-Face[29] 模型服务和位于云端的人脸特征距离计算服务(从 1 000 张人脸中找出与输入最相似的人脸). 其中, Ray 运行时使用 Ray Serve 部署 VGG-Face 模型, 并利用 Ray 中的任务抽象执行向量计算. Docker 运行时采用 TensorFlow Serving 部署 VGG-Face 模型, 并基于 Flask[30] 构建向量计算的 HTTP 服务.

实验结果如图 3 所示, 相较于仅使用 Docker 或 Ray 运行时, 采用多运行时部署应用分别可以降低 44% 和 34% 的平均延迟、42% 和 44% 的 P90 和 P95 尾延迟. 进一步分析发现, 与 Ray 相比, Docker 运行时将 VGG-Face 模型平均推理时延和 P90, P95 尾延迟降低至少 45%. 而与 Docker 相比, Ray 运行时在人脸特征距离计算方面的延迟降低了 53%. 因此, 为不同的负载类型选择合适的运行时可大幅提升应用性能.

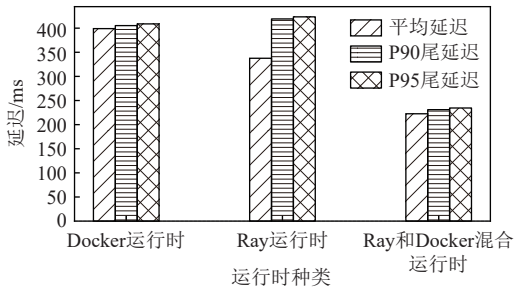


Fig. 3 Performance comparison of face verification applications using multiple runtimes and a single runtime

图 3 采用多运行时与单一运行时的人脸验证应用性能比较

多运行时可降低网络异构对应用性能的影响. 端边云应用的网络环境十分多样化, 而分布式运行时通常只适用于部分网络环境. 例如, 在数据中心场景中, Ray, K8s 等分布式运行时, 大多采用基于 TCP 协议管理数据平面和控制平面. 一些新型运行时, 如 Nu[31], 采用了需要百位数 Gbps 网络带宽的 RDMA 协议. 在端边云协同计算场景下, 物端之间以及物端和边缘之间的网络可靠性和带宽通常低于数据中心网络. 同时, 文献 [15] 指出, 在较差环境下, 基于 Wi-Fi 的丢包率可达到 10%. 为验证多运行时可降低网络异构对应用性能的影响这一观点, 本文在物端和边缘设备上部署了 Ray 分布式运行时和基于 UDP 的运行时, 并测试 2 种运行时在不同丢包率条件下执行人脸检测任务的延迟. 人脸图片大小为 5 MB, 网络带宽为 500 Mbps. 实验结果如图 4 所示, 基于 UDP 的运行时的任务执行延迟基本不受丢包率的影响, 而 Ray 运行时的任务执行延迟则增加了数十倍. 因此, 在复杂网络场景下构建应用时, 选择合适的运行时可显著降低应用的端到端延迟.

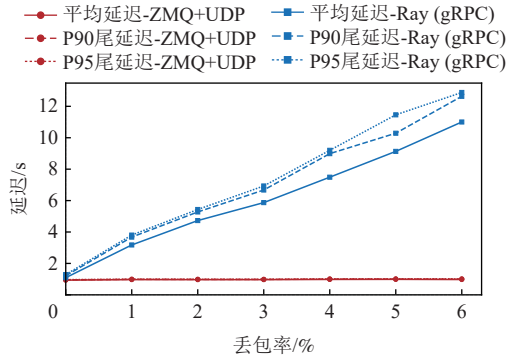


Fig. 4 Performance of Ray and the UDP-adopted framework under varying packet loss rates

图 4 Ray 和采用 UDP 的框架在不同丢包率下的性能

利用不同抽象层级的运行时, 可在满足用户安全隔离需求的条件下提升启动速度. 如表 1 所示, 各运行时为用户提供了不同层级的编程抽象、隔离能力和任务启动速度. 例如, VMware[32], PVE(Proxmox virtual environment)[33] 提供了虚拟机级别的运行时抽象, 隔离能力强, 但需加载操作系统内核, 任务启动速度通常为分钟级. T-REST 则允许用户直接提交函数, 并在运行时对应的进程中执行, 无需重复加载库, 因此任务启动延迟可降低至约 10 ms.

Table 1 Task Isolation and Start-Up Differences at Different Abstract Levels of Runtimes

表 1 不同抽象层级运行时的任务隔离与启动差异

抽象层级	隔离程度	任务启动速度	运行时
虚拟机	高	分钟级	VMware, PVE
容器	中	秒级	Docker, K8s
进程	低	百毫秒级	Ray, Dask
函数	最低	十毫秒级	SOCK, T-REST

物端设备和个人边缘设备通常非共享, 可使用较低隔离能力的进程或函数执行任务, 从而实现更快的启动速度. 在同一组织内共享的边缘设备中, 可采用中等隔离能力的容器执行任务, 以减少启动延迟.

2.2 自定义缩扩容和容错策略的优势

自定义缩扩容策略(custom-scaling-strategy, CSS)可显著降低应用的平均延迟和尾延迟. 现有分布式运行时的容错和缩扩容策略通常是确定的. 仅采用 1 种运行时会导致用户难以定制应用组件的执行方式, 从而降低应用整体性能. 本节对比了分别使用 K8s 系统和支持自定义缩扩容策略的容器运行时构建的应

用性能，支持了上述观点.

K8s 与自定义缩扩容策略容器运行时拥有的资源总量一致，且均预先部署 1 个实例，最多允许部署 2 个实例. K8s 系统采用非常激进的扩容策略：资源状态 metrics 采集周期设置为 12 s（metrics server 不支持设置低于 11 s 的采集周期），pod 数量同步周期设置为 500 ms（默认值为 15 s），initial readiness delay 设置为 500 ms（默认值为 30 s），扩容触发条件为 CPU 利用率超过 10%. 自定义缩扩容策略容器运行时的扩容触发条件为：服务并发处理的请求数量超过 3 个；客户端每 300 ms 发送 1 个服务请求，共发送 100 个请求. 结果显示，在自定义缩扩容策略下，自定义缩扩容策略容器运行时 4.5 s 完成扩容；然而 K8s 系统需 21 s 才完成扩容. 与 K8s 相比，自定义缩扩容策略容器运行时分别降低了 59% 的平均延迟以及 71% 的 P90 和 P95 尾延迟.

自定义容错策略可提升应用鲁棒性和性能. 首先，开发者在构建端边云应用时会直接使用单机运行时. 例如，在物端设备上使用 Python 运行时部署 Python 代码，在边缘设备上使用 Docker 部署容器化应用. 然而，这些单机运行时并不提供容错能力，为这些运行时提供容错支持可提升应用的鲁棒性. 其次，文献 [28] 的研究结果显示，支持开发者自定义容错策略可大幅降低因容错需求带来的执行开销和错误恢复开销，进而提升应用性能.

2.3 研究问题的形式化分析

在本文中，端边云协同应用的多运行时执行模型被定义为有向无环图 $G=(V,E)$. 其中，$V\subseteq R\times K\times E\times S$，$R$ 是所有运行时的集合，K 是所有任务的集合，E 是所有容错策略的集合，S 是所有缩扩容策略的集合. 顶点 $v=(r,k,e,s)\in V$ 的含义是：任务 k 在运行时 r 上执行且采用容错策略 e 和缩扩容策略 s. $E\subseteq V\times V$，表示顶点之间的创建关系：如果任务 k_1 在运行时 r_1 上执行时创建了任务 k_2，并且任务 k_2 在运行时 r_2 上执行，则对应于一条从 (r_1,k_1,e_1,s_1) 指向 (r_2,k_2,e_2,s_2) 的边.

本文旨在为用户构建的上述有向无环图 G 提供系统级支持，包括部署、管理、容错以及缩扩容. 具体而言，需要解决以下 3 个关键问题：

1）统一接入和使用问题

$$\exists I,\forall r\in R,\forall k\in K,\\ ManExec(r,k)\Rightarrow AutoExec(I(r),I(k)). \tag{1}$$

式（1）的含义是提供一种统一的方法 I，使得对于任意任务 k 和运行时 r，如果任务 k 可通过人工方式在 r 上执行（即 *ManExec*），则在方法 I 的封装下，可自动在 r 上执行 k（即 *AutoExec*）.

2）统一生命周期管理问题

$$\exists L,(\forall G,ManMng(G)\Rightarrow AutoMng(L(G)))\wedge\\ (\forall v\in V,ManMng(v)\Rightarrow AutoMng(L(v))). \tag{2}$$

式（2）的含义是提供一种统一的生命周期管理方法 L，使得对于任意的应用 G，如果可通过人工方式对应用整体进行生命周期管理，包括启动、暂停和终止等（即 *ManMng*），那么在方法 L 的封装下，也可自动实现对应用整体的生命周期管理（即 *AutoMng*）. 此外，每个应用组件的生命周期管理也应满足相同的要求.

3）任务级的自定义容错和缩扩容策略问题

$$\exists M,\forall v=(r,k,e,s)\in V,\forall t_1,\exists t_2>t_1,\\ Failed(v,t_1)\Rightarrow Recovered(k,e,t_2). \tag{3}$$

式（3）的含义是提供一种任务级自定义容错支持机制 M，使得对于任意的任务 k，当其在时刻 t_1 出错后，总会在某个大于 t_1 的时刻 t_2，运用策略 e 进行错误恢复. 缩扩容问题的形式化描述与此类似，只需将错误恢复策略替换为缩扩容策略.

上述系统级支持涵盖了从应用开发到执行的整个流程. 在初始条件下，系统中仅包含运行时集合 R. 用户通过编写应用的业务逻辑得到任务集合 K，并提供每个任务与运行时的配对需求信息 i、容错策略 e 和缩扩容策略 s. 系统依据信息 i，将任务 $k\in K$ 自动部署到满足需求的运行时 $r\in R$，并记录相应的缩扩容策略和容错策略，从而完成图 G 的初次构建. 在应用的执行过程中，如果创建了新任务或进行了容错、缩扩容操作，系统将重复上述步骤，动态更新图 G.

3 网程系统的设计

本节讨论了网程系统设计的 5 个基本需求，引入了网元和网程抽象以支持多运行时应用的生命周期管理，提出了虚运适配器与虚运 API 以支持多运行时统一接入和使用，并介绍了网元级自定义容错和缩扩容策略的支持机制. 此外，还探讨了网程系统的编程方式，并通过应用实例说明网程的执行过程.

3.1 概 述

网程系统需满足多运行时应用的统一管理、兼容现有生态、任务级自定义容错策略 3 个功能需求，以及性能保持、任务级自定义缩扩容策略 2 个性能需求.

网程系统提供网元与网程以及虚运适配器与虚运 API 这 2 类抽象，分别应对应用组件与应用以及运

行时接入与使用的异构性, 支持了多运行时应用的统一管理, 包括各类运行时的统一接入和使用以及应用生命周期的管理.

网程系统采用旁观式设计支持了生态兼容和性能保持, 使开发者可直接复用各运行时已有的库和代码, 并实现与人工构建以及部署与管理的应用相当的性能. 具体体现在 2 个方面:

1)网程系统不引入新的运行时, 而是通过运行时适配器在目标运行时上直接执行用户代码, 支持用户直接使用现有生态中的代码资源, 满足了生态兼容的需求. 如果引入新的运行时或对可使用的运行时进行限制, 用户则需要修改应用代码, 阻碍对现有生态的利用. 例如, Ray 是专为 Python 设计的, 难以直接利用非 Python 的生态资源.

2)将任务部署到相应的运行时后, 网程系统默认不介入任务的正常执行流程, 以保持任务性能和人工管理与部署的一致. 如果网程系统位于任务的正常执行路径中, 则会干扰或直接降低应用性能. 例如, 边车会带来代理转发开销.

网程系统引入了所有权机制, 支持网元级别的自定义缩扩容和容错策略. 每个网元可配备独立的缩扩容管理者和容错管理者, 负责自定义策略的执行. 如果应用未指定策略, 系统将自动选择默认的缩扩容和容错管理者.

图 5 给出了网程系统的模块图. 系统由中心化的网程管理器、分布式的网元管理器、运行时调度器以及众多的运行时构成. 每个运行时须基于虚运适配器实现各自的运行时适配器, 以接入网程系统. 提交的任务依次通过运行时 API、虚运 API、虚运适配器、运行时适配器, 最终在目标运行时上执行. 网程管理器负责记录应用对应的网程信息. 网元管理器记录其所在设备的所有网元的元数据. 运行时调度器接收虚运适配器上报的运行时状态, 并根据资源需求为任务分配合适的运行时. 网程系统采用最近优先使用的调度策略, 优先选择已部署该网程的机器, 如果资源不足则随机选择满足资源需求的机器. 如图 5 中应用任务的“网元”部分所示, 任务也可使用网程系统的功能, 例如提交新的子任务.

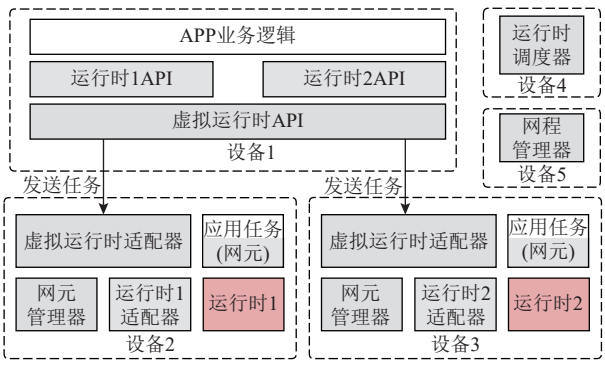


Fig. 5 Diagram for Grip system modules

图 5 Grip 系统模块图

3.2 网元与网程抽象及其管理

定义 1. 网元. 网元是网程系统中最小可管理单元, 是各种运行时上正在执行的任务的统一抽象. 网元的控制块(结构体)包括网程名、网元名、签名、同构名、运行时自定义数据、状态以及功能、容错和缩扩容管理者字段. 网元定义了必须实现的接口, 包括终止、健康检查、配置查询、结果查询、通信、缩扩容管理和容错管理.

定义 2. 网程. 网程是运行中的分布式应用, 它由网元及网元间的创建关系构成. 如果网元 A 创建了网元 B, 则称 A 是 B 的父网元. 创建关系的唯一性和单向性, 使得网程是由网元构成的一棵树状结构, 并被保存在网程控制块中.

网程与网元均采用全局唯一标识符(universally unique identifier, UUID)来命名以确保其唯一性, 并支持通过名字进行生命周期管理. 网程系统通过网元签名支持错误恢复后应用状态的一致性. 网元签名在其兄弟网元中是唯一的. 因此, 当网元采用重执行策略进行错误恢复时, 恢复后的网元可使用该签名获取一致的网元名, 以实现应用状态的一致性. 用户可通过同构名获取对应的同构网元, 以支持需要缩扩容的场景. 同构性由应用定义, 如功能相同的服务实例. 如图 6 所示, 网元的默认功能管理者是启动该网元的运行时适配器, 默认的容错和缩扩容管理者是启动其父网元的运行时适配器. 上述设计可减少用户自定义 3 种管理者的需求. 具体原因如下: 1)网元是由所在运行时的运行时适配器启动的, 由它来负责网元终止、配置查询、结果查询等, 可利用已具备的上下文简化管理流程. 2)系统的 2 类组件必然拥有网元对应的任务信息. 其一是网元的父网元, 其二

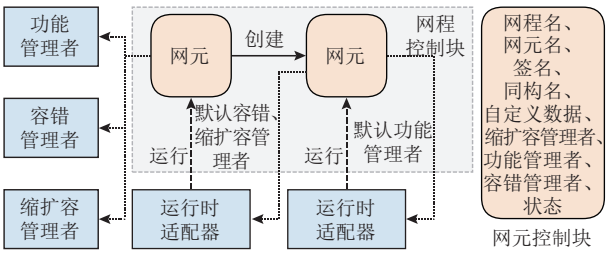


Fig. 6 Control block of Grip and Griplet

图 6 Grip 与 Griplet 控制块

是启动网元的运行时适配器. 对于单机运行时, 网元的出错通常意味着其运行时适配器也发生故障. 因此系统选择父网元的运行时适配器作为网元的默认容错管理者. 为支持默认的错误恢复策略, 系统要求父网元将任务的自定义数据保存至本地网元管理器. 通过支持多个分布式的容错管理者, 降低了整体容错开销.

系统通过网程和网元管理器分别管理网程和网元的生命周期. 网程管理器负责存储、维护和查询网程状态, 并提供创建和终止网程、添加和更新网元、网元的缩扩容和容错 6 种功能. 网元管理器存储和维护本地多个网程的网元状态, 并与网程管理器交互. 网元的自定义数据保存在网元管理器中, 可降低网程管理器的负载压力. 具体而言, 应用首先会通过虚运 API 向网元管理器发送创建网元请求. 然后, 网元管理器从请求中解析出网程名、网元名、签名、同构名、3 类管理者的名字、所需运行时等信息, 并将这些信息保存至网元控制块中. 最后, 网元管理器会将网元的元信息发送给网程管理器, 网程管理器自动判断网元是否为初次创建, 并更新网程控制块. 当网元开始执行或终止时, 虚运适配器会通知网元管理器. 网元管理器会更新网元的状态并通知网程管理器.

应用可通过网元的统一接口执行健康检查、结果查询、终止等管理操作. 上述命令会经过网程管理器(可选)、网元管理器、虚运适配器, 最终由运行时适配器完成命令的执行. 我们将在 3.3 节讨论具体细节.

3.3 虚运适配器与 API

如 3.2 节所述, 各类运行时通过各自的运行时适配器提供任务执行和管理服务. 然而该方式并未解决运行时异构的问题, 无约束的接入会提升运行时使用和应用管理的复杂度.

网程系统通过引入虚运适配器和虚运 API 中间层支持各类运行时的统一接入与使用. 虚运适配器从生命周期角度规定了运行时适配器须实现的 8 个接口. 虚运 API 提供了与虚运适配器交互的 7 个接口. 网程系统以框架的形式提供了虚运适配器和虚运 API 的实现, 以及网元和网程管理所需的功能.

虚运适配器规定的接口涵盖了任务执行的全生命周期. 开始和结束是生命周期必须包含的 2 个阶段, 因此规定了提交接口和终止接口. 在分布式应用中, 为保证应用的鲁棒性和性能, 容错机制与弹性缩扩容策略是必须考量的关键因素. 因此规定了错误处理接口和缩扩容接口. 任务执行的核心目标通常在于获取其结果, 因此需提供结果获取接口. 任务的状态查询是任务交互、错误恢复的必要功能, 因此规定了配置查询接口和健康检查接口. 最后, 为确保系统的完备性, 规定了网元通信接口, 允许应用向网元传输任意二进制数据, 以支持其他个性化功能需求.

表 2 详细列出了运行时适配器需要实现的接口规范. 虚运适配器提供了上述接口的通用实现, 以支持接入、使用与生命周期管理语义的一致性. 同时, 虚运适配器预留了可由运行时适配器自定义的二进制字段, 以支持应用逻辑的多样性. 例如, 提交接口的参数包括网程名、网元名、错误恢复标识、容错管理者引用、缩扩容管理者引用、资源需求以及任务体. 其中, 任务体是由运行时 API 与适配器定义的二进制数据段, 以支持执行不同运行时的任务, 如执行 Python 脚本、启动容器等. 此外, 资源需求中也包含一个二进制段, 允许用户自定义资源类型. 虚运适配器还为每个网元提供了通信地址, 支持网元间通信, 并降低接入运行时的开发复杂度.

Table 2 Interfaces Specified by the Virtual Runtime Adapter that the Runtime Adapters Need to Implement

表 2 虚拟运行时适配器规定运行时适配器要实现的接口集

接口	输入	输出
提交	网程名、网元名、父网元名、运行时自定义任务体、3 种管理者(可选)、错误恢复标识、签名、同构名、资源需求	运行时名、网元名、3 种管理者、自定义数据、状态、网元通信地址
终止	网程名、网元名	是否成功
配置查询	网程名、网元名	运行时名、网元名、3 种管理者, 自定义数据、状态、网元通信地址
结果获取	网程名、网元名	运行时自定义数据
错误处理	网程名、父网元名、网元名	是否成功、3 种管理者
缩扩容	网程名、父网元名、网元名	新网元名、3 种管理者
网元通信	网程名、网元名、运行时或用户自定义数据	运行时或用户自定义
健康检查	网程名、网元名	是否健康

基于虚运 API, 运行时 API 须实现提交接口、指向网元的句柄(handle)以及句柄的方法. 具体说明如下: 1)运行时 API 的提交接口. 将该运行时的任务转换为二进制数据, 并指定网程名、网元名、错误恢复标识和资源需求等参数; 然后通过虚运 API 将任务提交请求发送到虚运适配器, 最终在目标运行时上执行; 虚运 API 的提交接口会返回一个网元句柄. 2)网元句柄及其方法. 虚运 API 提供了一套通用的句柄方法, 包括健康检查、终止、缩扩容、容错等. 除包含网元名、容错管理者引用、缩扩容管理者引用等

通用字段外, 句柄还包含运行时自定义的 *data* 字段. 运行时 API 可通过赋予 *data* 字段特定语义, 为网元句柄扩展新的方法. 例如, 在 Docker 运行时 API 中, *data* 字段被用于定义端口的映射关系, 并提供 *access* 方法访问容器内的服务. 表 3 详细列出了虚运 API 为运行时 API 所提供的接口集.

Table 3 Interfaces Provided by the Virtual Runtime API for Runtime API

表 3 虚拟运行时 API 为运行时 API 提供的接口集

接口	输入	输出
提交	必要参数: 运行时自定义任务体可选参数: 网程名、3 种管理者、资源需求、签名、同构名、错误恢复标识	是否成功、指向网元的句柄
句柄. 终止	无	是否成功杀死网元
句柄. 获取结果	无	运行时自定义数据
句柄. 配置查询	无	运行时名、网元名、3 种管理者、运行时自定义数据、状态
句柄. 缩扩容	无	是否成功、句柄列表
句柄. 容错	无	是否容错成功 (如成功会更新句柄)
句柄. 网元通信	方法名、运行时或用户自定义数据	是否成功、运行时或用户自定义数据

图 7 展示了使用运行时的典型执行过程. 在此过程中, 应用调用运行时 API 的接口 I_r, 例如提交、查询等. I_r将参数转换为虚运 API 相应接口 I_{vr}的参数格式, 然后调用该接口. I_{vr}随后向虚运适配器发送网络请求. 虚运适配器收到后, 调用运行时适配器中对应的接口 I_{ra}, 最终完成接口调用.

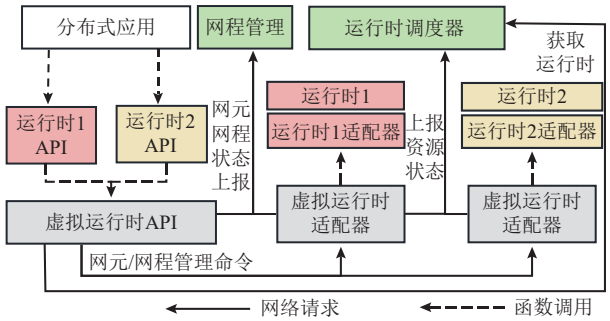


Fig. 7 Execution process of virtual runtime API and adapter

图 7 虚拟运行时 API 和适配器的执行过程

如图 7 所示, 虚运适配器和虚运 API 还提供了网程生命周期管理所需的各种功能, 包括资源状态上报、请求传输、启动运行时适配器服务、网元的创建与更新等. 因此, 在开发运行时适配器和运行时 API 时, 开发者只需专注于业务逻辑, 如运行时的任务格式、与外部系统关联的管理操作等.

3.4 网元级自定义容错和缩扩容策略

每个网元可设置单独的容错和缩扩容管理者. 这 2 类管理者可采用应用自定义的策略, 以支持网元级自定义容错和缩扩容策略.

网程系统提供自动容错的支持. 当任意网元发现其他网元出错后, 可向其本地的网元管理器发送错误恢复请求. 该请求仅需包含网程名、网元名和功能管理者名 3 项内容. 网元管理器接收到请求后会通知网程管理器进行错误处理. 系统采用递归式错误恢复机制. 首先, 网程管理器会检查请求中的功能管理者名是否与其保存的一致, 如不一致则直接返回当前的网元配置, 避免重复的错误恢复. 否则会根据出错网元的容错管理者名, 向容错管理者发送错误恢复请求, 如恢复成功则返回新的网元配置, 并更新功能管理者名; 如果失败且是因为容错管理者失活, 则网程系统会查询其父网元的容错管理者名, 递归进行错误恢复.

图 8 展示了网元及其父网元同时出错时, 网程系统进行错误恢复的示例, 包含 5 个步骤:

①网程管理器接收到网元 3 出错的消息.

②网程管理器尝试向网元 3 的容错管理者发送错误恢复请求, 该请求失败.

③网程管理器查询网元 3 的父网元(网元 2), 并向网元 2 的容错管理者发送错误恢复请求.

④假设容错管理者的自定义容错策略为重执行, 则会重新提交网元 2.

⑤在网元 2 的重执行过程中, 可通过代码中的签名获取与出错网元一致的名字, 并重新创建网元 3.

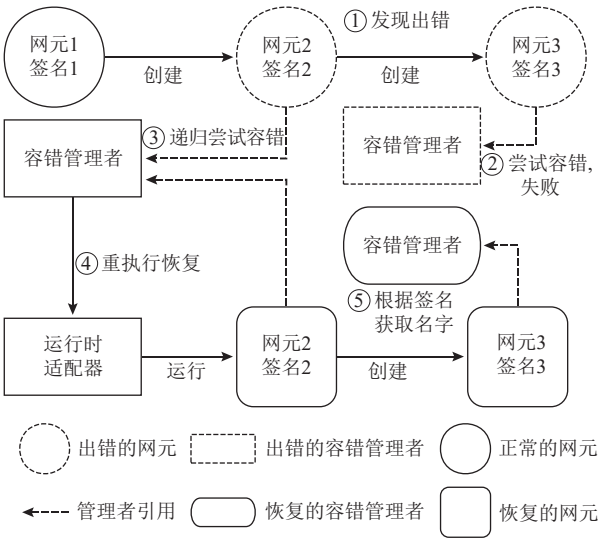


Fig. 8 Support mechanism of custom fault-tolerant strategy at network element level

图 8 网元级自定义容错策略的支持机制

至此完成错误恢复.

默认情况下, 网元的容错管理者是启动其父网元的运行时适配器, 将图 8 中的容错管理者替换为对应的运行时适配器, 即可得到默认容错机制的示例.

类似地, 网程系统提供自动缩扩容的支持. 当任意网元发现某网元不满足保质要求时, 可通过网程系统自动通知该网元的缩扩容管理者执行自定义或默认的缩扩容策略. 此外, 网程系统提供了同构机制以支持缩扩容. 开发者可赋予一组网元相同的同构名, 将它们设为同构网元, 以支持负载均衡和快速重试等机制. 例如网元可创建一组同构网元, 并作为它们的代理, 将负载均衡地分配到同构网元. 当网元过负载时, 可将请求发送到空闲的同构网元, 实现快速重试.

3.5 编程方式

网程系统不改变用户使用运行时的方式, 通过提供编程接口, 支持用户在单个应用中采用多种运行时. 网程应用的典型开发流程如下: 1)编写各个运行时上执行的任务. 2)根据业务逻辑, 确定任务之间的创建关系. 通过相应的运行时 API, 编写任务的提交代码, 并可指定其功能管理者、错误管理者和缩扩容管理者. 3)编写根任务代码, 并启动根任务. 根任务执行过程中, 会依据创建关系, 提交其子任务; 类似的, 子任务也可提交自己的子任务. 上述过程递归进行, 直至完成所有任务的执行.

应用通过运行时 API 的 *submit* 接口提交任务. 任务除了包含待执行的代码和输入参数外, 还可指定计算资源需求(如 CPU、内存)、网元名、网程名、任务签名及同构名. 用户可仅提供代码和输入参数, 其余参数由运行时 API 和虚运 API 自动补全.

任务提交后, 运行时 API 会返回一个指向相应网元的句柄. 用户可通过该句柄对网元进行功能、缩扩容和容错管理. 该句柄支持 5 种默认的功能管理操作: 获取网元执行结果的 *get* 操作、与网元交互的 *invoke* 操作、获取网元配置的 *query* 操作、进行健康检查的 *health* 操作以及终止网元的 *kill* 操作. 该句柄通过 *errorReport* 操作支持网元的容错, 并返回恢复后网元的功能管理者名, 随后通过 *query* 操作更新句柄的元数据.

3.6 应用实例

图 9 展示了基于网程系统的人脸验证应用代码示例. 该应用分别使用 Docker 运行时和 Ray 运行时执行人脸特征向量提取的 VGG-Face 模型服务和人脸特征向量对比的验证服务.

```
/*根任务的代码：首先提交使用Docker 运行时的VGG-Face模型服务任务，返回其vggFaceH句柄；然后提交验证服务任务，参数包括vggFaceH句柄和参考人脸地址，返回verifyH句柄；最后通过调用verifyH的access方法验证待验证人脸*/
①vggFaceH=dockerAPI.submit(imageName, sig1, iso1,
  internalPorts=["8501"], cpu=10, mountFolder);
②verifyH=rayAPI.submit(codeUrl, sig2, iso2,
  args=[vggFaceH, faceUrl], portNum=1);
③verifyH.access(portIndex=0, "POST", "/verify",
  faceData);
/*验证服务：通过vggFaceH获取特征，使用Ray计算人脸距离*/
①port=griplet.ports[0];
②vggFaceH=griplet.args[0], faceUrl=griplet.args[1];
③represent= vggFaceH.access("8501", "POST",
  "/representation", requests.get(faceUrl).content);
④@ray.remote;
⑤def verify(faceData);
  /*使用vggFaceH获取人脸特征，计算与原始人脸的距离*/
  /*运行http服务器，在"/verify"路径对应的函数中调用verify*/
```

Fig. 9 Grip application of face verification based on VGG-Face model

图 9 基于 VGG-Face 模型的人脸验证的 Grip 应用

在根任务代码中, 应用通过 *dockerAPI* 提交 VGG-Face 模型服务任务并返回网元句柄(*vggFaceH*). 该任务使用 TensorFlow-Serving 镜像, 并指定需要公开的服务端口. 随后, 通过 *rayAPI* 提交验证服务任务, 其参数包括 *vggFaceH*、参考人脸照片地址和服务端口信息. 同样的, 验证服务任务提交后, 返回验证服务句柄(*verifyH*). 最后, 通过 *verifyH* 发送人脸验证请求, 以判断请求中的人脸和参考人脸是否为同一人.

在验证服务代码中, 应用使用网程系统提供的库获取分配的端口、模型服务句柄(*vggFaceH*)和参考人脸照片地址. 然后通过 *vggFaceH* 的 *access* 接口获得参考人脸照片的特征向量. 最后, 启动 HTTP 服务, 该服务接受图片输入, 并执行 *verify* 函数(Ray 任务), 该函数通过 *vggFaceH* 的 *access* 接口获取输入图片中人脸的特征向量, 并计算其与参考人脸特征向量的距离.

图 10 展示了人脸验证应用的执行过程, 包括 5 个阶段: ①根任务向网程管理器发送创建网程的请求. ②根任务向运行时调度器发起请求, 以申请 Docker 和 Ray 运行时. ③根任务向 Docker 运行时适配器提交 VGG-Face 人脸特征提取服务任务; 适配器收到任务后, 在 Docker 运行时上执行该任务, 并通知本地网元管理器创建网元. 本地网元管理器将网元的元信息发送至网程管理器, 同时在网元的容错管理者中保存错误恢复所需的数据. ④提交并执行人脸特征向量对比的验证服务任务, 该过程与提交 Docker 任

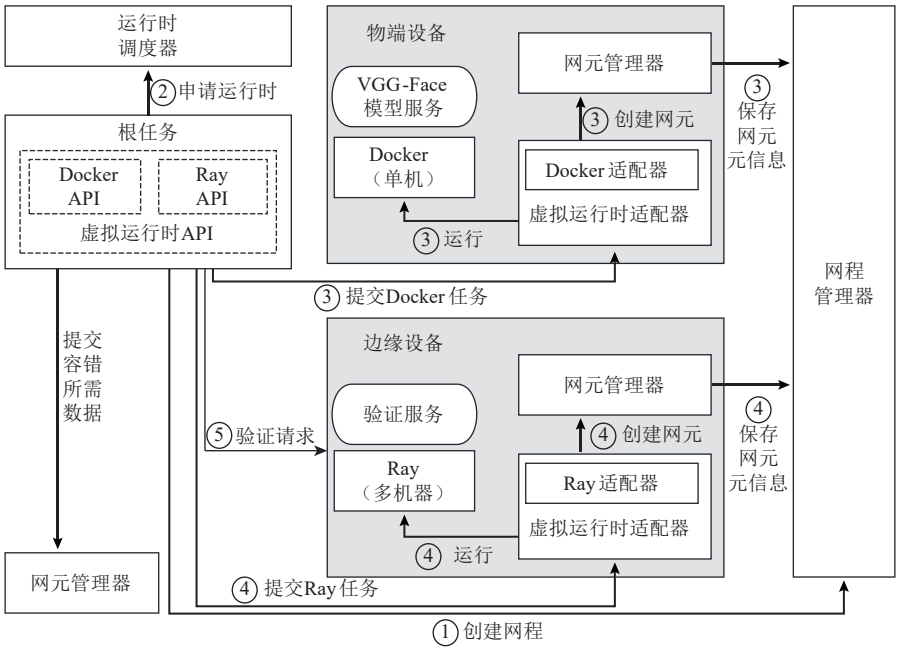


Fig. 10 Execution process of face verification for Grip application

图 10 人脸验证 Grip 应用的执行过程

务类似，不再赘述. ⑤根任务通过验证服务网元的句柄发起验证请求；验证服务在收到请求后，发起 Ray 的远程任务. 该远程任务利用 VGG-Face 模型服务网元句柄，发起人脸特征向量提取请求，并计算参考人脸特征向量与输入图片中人脸特征向量的距离，最终返回验证结果.

4 实　　验

本节通过基准测试和端边云协同应用实验，分析了网程管理操作的开销，比较了网程系统与单一运行时系统的性能差异，以及网程应用与全人工构建应用的性能差异. 实验旨在回答 4 个问题：

1)网程系统的生命周期管理开销是多少(见 4.1 节).

2)使用多种抽象层级的运行时，是否能在突发负载时加速扩容，从而降低端到端延迟(见 4.2 节).

3)在端边云的异构网络环境下，与单一运行时相比，使用多运行时是否能有效降低应用的整体延迟(见 4.3 节).

4)在端边云协同场景下，通过多运行时的统一管理，网程是否能支持自定义缩扩容策略并降低延迟，以及支持自定义容错策略以实现快速容错，并与人工部署管理的应用性能相当(见 4.3 节).

4.1 基准测试

基准测试程序用于评估网程系统各操作的开销. 具体而言，实验测试了提交开销、获取空任务执行结果的延迟、查询网元配置的延迟、容错开销、缩扩容开销、向网元发送空消息的通信开销以及终止空任务的开销. 其中，提交开销是指从提交一个空任务到其成功运行的延迟；容错开销是指从报告网元错误到重新提交执行空任务的延迟；缩扩容开销是指从报告网元扩容需求到执行第一个扩容的空任务的延迟.

实验结果如图 11 所示，网程系统的任务提交开销平均约为 9 ms. 获取结果、配置查询和网元通信开销均小于 2 ms. 终止开销约为 4 ms. 容错和缩扩容开销约为 14 ms. 由于容错、缩扩容和提交操作涉及运行时适配器、网元管理器和网程管理器之间的网络通信，因此其开销相对较大. 相比之下，先进的 Ray 分布式框架提交空任务到获取结果开销约为 6.2 ms，网程系统完成相同功能的开销约为 9.8 ms. 这一差异源于网程原型系统基于 Python 的 Flask 框架构建，而

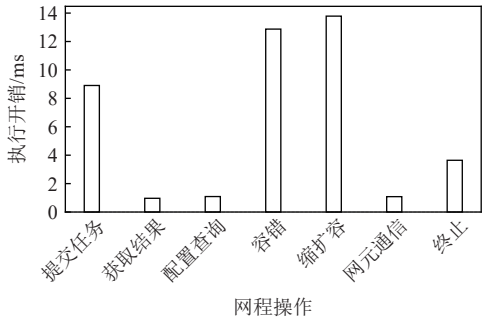


Fig. 11 The overhead of Grip operations

图 11 Grip 操作的开销

Ray 则基于 C++和 gRPC 构建. 未来工作将采用更高效的编程语言和服务框架, 以优化网程系统的通信性能.

4.2 家庭边缘语音助手应用测试

本节通过边缘语音助手应用, 分析了采用多种抽象层级的运行时构建的应用性能. 在满足用户安全和隔离需求的前提下, 此方法显著提升了任务启动速度, 并降低了 31%~54% 的平均端到端延迟, 同时扩容阶段减少了 54%~65% 的服务请求延迟.

4.2.1 应用描述

如图 12 所示, 边缘语音助手应用由物端和边缘端 2 部分构成. 物端采集音频, 将采样频率调整至 16 kHz, 将声音数据发送至语音助手接入服务并等待结果. 当物端接收到回答文本后, 将其合成为语音. 边缘端包括语音助手接入服务和语音转文字服务. 语音助手接入服务的功能包括接收音频, 将音频发送到语音转文字服务, 根据识别出的文本生成回答以及确定语音转文字服务的实例数量.

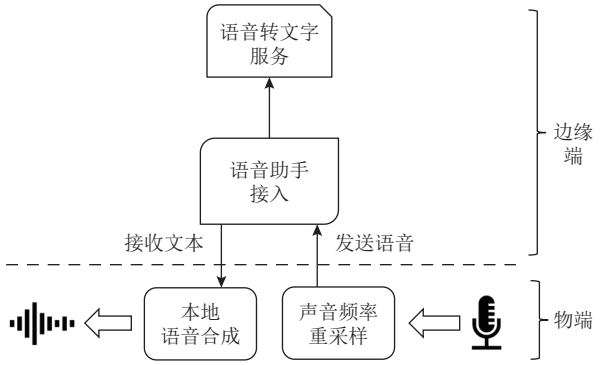


Fig. 12 Edge voice assistant application

图 12 边缘语音助手应用

物端使用 festival[34] 进行语音合成, 并利用 SciPy 库进行音频重采样. 边缘端使用 Wav2vec2[35] 模型执行语音转文字.

4.2.2 实验设置

实验采用 2 台服务器模拟家庭场景中的端边设备, 其中 1 台服务器模拟物端, 另 1 台服务器模拟边缘端. 每台服务器配备 16 核 CPU、32 GB 内存, 并通过 10 Gbps 网络互联. 为模拟端边云网络环境, 基于 802.11ac 的单流带宽标准[36], 使用 Linux 中的 tc 工具将带宽限制为 500 Mbps.

网程版语音助手应用采用 Python 运行时和基于 fork 机制的函数级无服务器运行时(简称函数级运行时)实现. 其中, 端侧使用 Python 运行时, 边缘侧语音助手接入服务和语音转文字服务均采用函数级运行时. 实验中对比了仅使用 Docker 运行时和仅使用 Ray 运行时的语音助手应用.

网程、Ray 和 Docker 这 3 个版本的应用采用一致的扩容策略: 在缺乏立即可用的服务时进行扩容, 服务实例数量上限设定为 4. 实验开始时, 3 个版本均部署 1 个实例, 并限制每个实例使用 3 个 CPU 核. 任务请求的平均发送频率为每秒 6 次, 遵循泊松分布, 且请求的持续时间为 10 s, 以模拟突发负载场景.

实验假设家庭边缘服务器是用户可控且可信环境, 因此采用函数级运行时可满足安全隔离需求. 该运行时能够通过 fork 技术从已部署服务生成新服务. 这种技术已被应用于诸如 Pagurus[25], SOCK[12] 等运行时系统, 可大幅降低服务的启动开销.

4.2.3 实验结果

图 13 展示了不同版本边缘语音助手的端到端平均延迟、P90 尾延迟和 P95 尾延迟. 实验结果表明, 网程应用通过混合使用函数级运行时和 Python 运行时, 相比于仅使用 Ray 或 Docker 运行时, 分别降低了 54% 和 31% 的平均延迟、44% 和 25% 的 P90 尾延迟, 以及 44% 和 25% 的 P95 尾延迟.

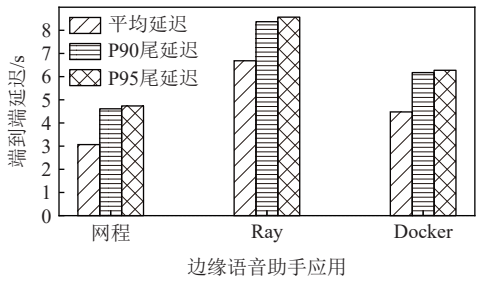


Fig. 13 End-to-end latency of different versions of edge voice assistant applications

图 13 不同版本的边缘语音助手应用端到端延迟

端到端延迟主要由执行延迟和排队延迟组成. 函数级运行时能够实现快速扩容和提高执行并发度, 从而有效降低排队延迟. 图 14 进一步展示了一次实

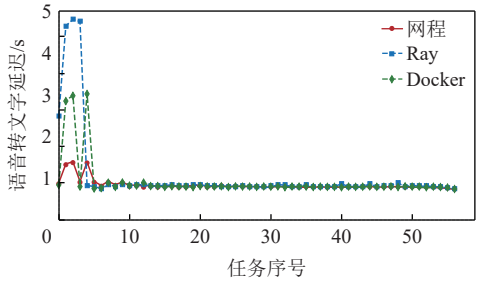


Fig. 14 Latency during the speech-to-text transformation phase in edge voice assistant applications

图 14 边缘语音助手应用中语音转文字阶段的延迟

验中每个请求的语音转文字的延迟. 结果显示, 在扩容阶段, 网程应用的语音转文字服务延迟相较于 Ray 和 Docker 版本分别减少了 65% 和 54%.

4.3 端边云协同寻人应用测试

本节通过端边云协同寻人应用, 在良好和存在丢包的网络环境中, 分别测试了网程系统的性能, 并与仅使用 Ray 运行时、Ray 和 K8s 混合使用(Ray+K8s)以及全人工构建的多运行时比较. 实验结果显示, 基于网程系统, 为应用的不同组件选择其合适的运行时并采用自定义缩扩容策略, 大幅提升了应用的性能并实现快速自动容错. 与仅使用 Ray 和 Ray+K8s 混合的运行时相比, 网程系统减少了 22%~78% 的延迟, 并与全人工构建的应用性能相当.

4.3.1 应用描述

寻人应用依托于人脸验证技术, 通过端边云协同的方式, 实时检测摄像头拍摄的图像中是否出现待寻人脸. 如图 15 所示.

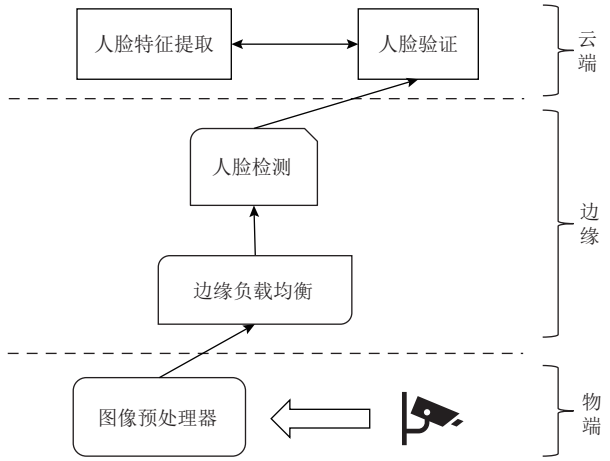


Fig. 15 People-finder application based on things-edge-cloud collaboration

图 15 基于端边云协同的寻人应用

该应用包括端侧图像采集、边缘预处理及云端计算分析 3 个部分, 各部分的实现如下所述.

1)端侧. 物端摄像头采集视频, 并将视频帧率由 30 fps 降至 10 fps. 随后, 利用 OpenCV 将视频帧压缩为 jpg 格式(压缩后的图像大小约为 1 MB). 最后将图像发送至位于边缘的负载均衡器.

2)边缘侧. 边缘负载均衡器接收到图像后, 将其分发至多个人脸检测器. 人脸检测器采用 MTCNN[37] 算法执行人脸检测, 裁剪出人脸图片, 并将其发送至云端的人脸验证器.

3)云端. 在初始化阶段, 人脸验证服务下载待寻人脸的参考图片, 并将其发送至人脸特征提取模型服务, 得到该人脸的特征向量. 随后, 验证服务持续接收来自人脸检测器的图像, 并将其发送至人脸特征提取模型服务以提取特征向量. 然后计算与参考人脸特征向量之间的距离, 从而判断是否匹配. 人脸特征提取模型服务基于 VGG-Face 模型构建, 其输入为任意大小图片, 输出为人脸特征向量.

4.3.2 实验设置

实验采用 8 台服务器模拟端边云设备, 其中 1 台用于模拟物端, 2 台用于模拟边缘端, 5 台用于模拟云端. 每台服务器配备 16 核 CPU、32 GB 内存, 并通过 10 Gbps 的带宽互连. 为模拟真实的端边云的网络环境, 实验利用 Linux 的 tc 工具进行网络配置. 参考 802.11ac 的单流带宽[36], 将物端和边缘端之间的带宽设定为 500 Mbps, 单向延迟为 5 ms, 并分别测试了物端网络丢包率为 0%, 1%, 3%, 5% 的 4 种情况. 边缘和云端之间的带宽设置为 100 Mbps, 丢包率为 0%, 单向延迟设置为 10 ms(参考了实验室南京和北京服务器之间的双向延迟约为 20 ms). 在边缘和云端内部, 网络带宽设为 10 Gbps, 且丢包率为 0%.

实验采用 4 种不同方式构建寻人应用.

方式 1: Ray. 仅使用 Ray 运行时构建, 其中人脸特征提取模型服务采用 Ray Serve(Ray 原生提供的模型服务框架)实现.

方式 2: Ray+K8s. 人脸特征提取模型服务采用 TensorFlow Serving 实现, 并部署于 K8s(版本 v1.30.1, 发布于 2024 年 5 月). 其余部分采用 Ray 运行时实现.

方式 3: 网程. 基于网程系统构建. 应用同时使用 Python+UDP 运行时、Docker 运行时和 Ray 运行时. 具体而言, 端侧图像预处理器、边缘侧的负载均衡和人脸检测器使用 Python+UDP 运行时构建, 物端和边缘端采用 UDP 协议通信(实验使用的传输协议见附录 A). 人脸验证服务采用 Ray 运行时构建. 人脸特征提取模型服务使用 Docker 运行时构建, 并采用自定义的缩扩容策略.

方式 4: 全人工构建. 应用各组件采用与网程版本一致的运行时, 但由人工进行部署和管理.

所有版本的应用均使用相同的计算资源. 实验开始时, 均部署 1 个边缘负载均衡、10 个人脸检测器实例和 1 个人脸验证服务实例.

在网程版本的应用中, 自定义缩扩容策略为: 当人脸验证服务检测到所有的人脸特征提取模型服务的并发请求数均超过 5 时, 发送扩容请求. 实验为 K8s 设置了一种激进的扩容策略: 当 CPU 利用率超过 25% 时触发扩容, 资源状态 metrics 获取周期设置

为 12 s(metrics server 不支持设置 11 s 及以下的周期)，pod 数量同步周期设置为 500 ms(默认为 15 s)，initial readiness delay 设置为 500 ms(默认为 30 s).

4.3.3 实验结果

图 16~18 展示了不同物端丢包率条件下，分别使用网程系统、Ray 和 Ray+K8s 实现的寻人应用性能表现. 由于 Ray 运行时采用 gRPC 进行通信，在丢包的端边环境下，有效通信带宽显著下降，导致应用延迟迅速增加. 相较之下，基于网程系统实现的寻人应用，在端边的任务中采用了支持 UDP 的运行时，因此其性能受丢包率的影响较小. 在丢包率为 1% 时，基于网程系统的应用在平均延迟以及 P90 和 P95 尾延迟方面，与 Ray 和 Ray+K8s 相比，分别降低了 60%, 51%, 49% 和 34%, 26%, 22%. 当丢包率升至 3% 时，基于网程系统的应用在这些指标上与 Ray 和 Ray+K8s 相比，分别减少了 69%, 65%, 65% 和 57%, 62%, 62%. 当丢包率达到 5% 时，基于网程系统的应用在平均延迟以及 P90 和 P95 尾延迟方面，与 Ray 和 Ray+K8s 相比，分别下降 77%, 78%, 78% 和 74%, 78%, 78%.

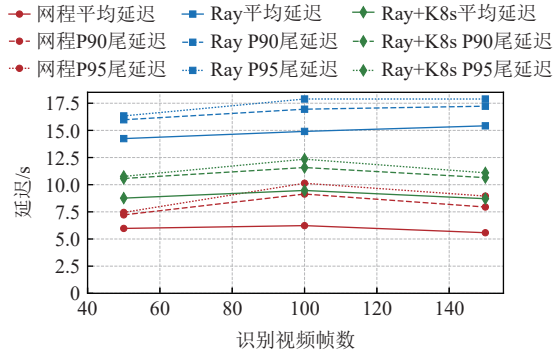


Fig. 16 End-to-end latency of single frame image (packet loss rate of things is 1%)

图 16 单帧图像端到端时延（物端丢包率为 1%）

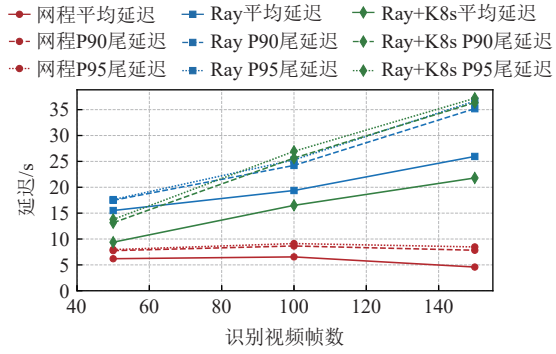


Fig. 17 End-to-end latency of single frame image (packet loss rate of things is 3%)

图 17 单帧图像端到端时延（物端丢包率为 3%）

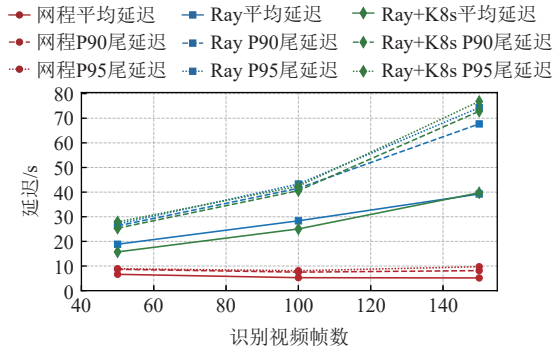


Fig. 18 End-to-end latency of single frame image (packet loss rate of things is 5%)

图 18 单帧图像端到端时延（物端丢包率为 5%）

考虑到端边云网络质量对应用性能的显著影响，为进一步探究自定义缩扩容策略对应用性能的影响，我们在无丢包的网络环境中进行了实验. 除物端丢包率设置为 0 外，其余实验配置和步骤保持不变. 图 19 展示了在无丢包网络中，3 个版本寻人应用的单帧图片处理的端到端延迟. 实验结果表明，网程版本相较于 Ray 和 Ray+K8s 版本，单帧图像的平均端到端处理延迟分别降低了 66% 和 37%, P90 尾延迟分别降低 69% 和 50%, P95 尾延迟分别降低 71% 和 55%.

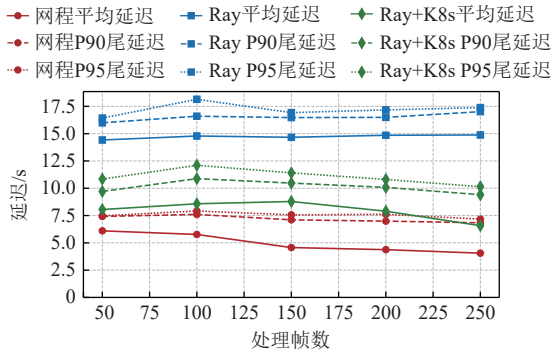


Fig. 19 End-to-end latency of single frame image (packet loss rate of things is 0%)

图 19 单帧图像端到端时延（无丢包）

图 19 显示，随着处理帧数的增加，平均处理延迟呈现下降趋势. 这是由于当扩容完成后，人脸特征提取的推理时间减少，进而导致平均延迟降低. 进一步分析发现，仅使用 Ray 运行时应用性能较低的原因是其 CPU 利用率偏低，即便在部署 8 个模型实例后，请求高峰期的 CPU 资源利用率也仅约为 40%; 相比之下，在 K8s 部署 2 个模型实例时，请求高峰期的 CPU 利用率可以接近 100%.

为了深入探究缩扩容策略对性能的影响，我们分析了单张图片人脸特征提取延迟随时间的变化. 图 20 展示了在 250 帧负载下，寻人应用的 3 种实现方式中，每张人脸图像的特征提取延迟随时间的变化. 实验结果表明，自定义的缩扩容策略显著降低了

模型推理的延迟. 与 K8s 的激进扩容设置相比, 自定义缩扩容策略使推理延迟进入到稳定状态的速度提升了 83%, 同时将延迟降低了 46%. Ray 运行时应用的延迟则数倍高于网程应用. 其他负载条件, 如 50 帧、100 帧的实验结果也呈现类似趋势. 如图 20 所示, 网程系统在约 120 帧时将延迟降低到稳定水平, 但网程应用在前 10 帧时即发出了扩容请求. 然而, 由于容器启动延迟较高, 即便发出扩容命令也无法立即提供可用服务, 从而阻塞了后续请求的执行. 此外, 由于实验使用了真实的街景视频, 前 100 帧中人脸较多, 推理服务调用更频繁, 这也导致了初期的推理延迟较高.

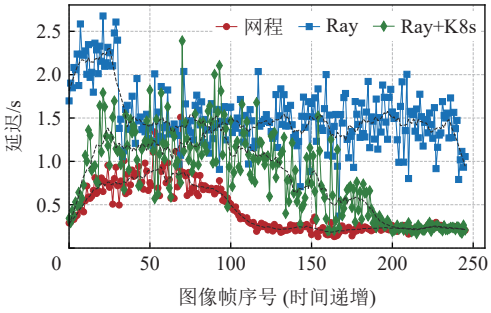


Fig. 20 Trend of single face feature extraction latency varying with the time

图 20 单次人脸特征提取延迟随时间变化趋势

图 21 展示了基于网程系统和全人工构建的寻人应用的性能. 全人工构建版本采用与网程版本相同的运行时, 但通过人工进行部署和管理. 在无丢包的端边云网络条件下, 进行了 10 轮测试. 结果显示, 两者性能无显著差异, 平均延迟差异不超过 3%.

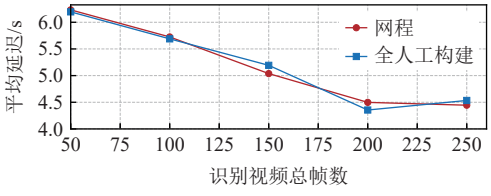


Fig. 21 Performance comparison of Grip version and the total manual construction version of the people-finder application

图 21 寻人应用 Grip 版本与全人工构建版本的性能比较

实验还验证了网程系统在支持自定义容错策略方面的有效性. 在应用执行过程中, 模拟了故障场景, 终止了边缘负载均衡所在节点上的所有组件, 包括边缘负载均衡服务和人脸检测实例. 得益于网程系统的支持, 通过重新提交边缘负载均衡任务和失效的人脸检测实例, 系统成功实现了错误恢复. 图 22 展示了应用出错与正常执行情况下的性能对比. 由于故障导致一半的人脸检测实例失效, 而这些实例依赖于初始化开销较大的 TensorFlow 库, 重新执行时需要数秒来重新加载这些库和模型. 这不仅引发了任务的阻塞, 还导致了资源争用, 从而增加了错误恢复后应用的端到端延迟. 结果表明, 网程应用在 400 ms 内完成了从错误发现到边缘负载均衡器重新执行的全过程.

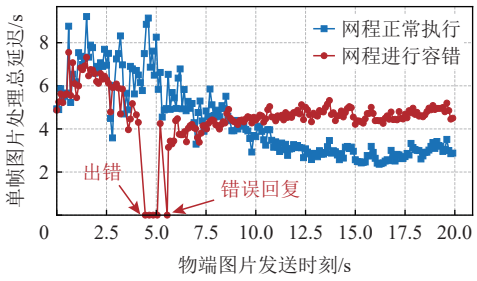


Fig. 22 Error recovery performance of the people-finder application

图 22 寻人应用出错恢复性能

5 结 论

本文提出并实现了网程系统, 通过解耦应用的生命周期管理和执行, 支持端边云协同应用统一使用多种运行时. 网程系统通过引入网程和网元抽象, 实现了多样应用的统一管理; 通过引入虚拟运行时适配器和虚拟运行时 API, 实现了多种运行时的统一接入和使用; 基于所有权机制, 允许应用为每个任务设置不同的容错和缩扩容管理者, 以支持网元级自定义容错和缩扩容策略.

端边云协同场景中的性能测试结果显示, 与使用 Ray, Ray+K8s, Docker 运行时相比, 在网络丢包的条件下, 端到端平均延迟、P90 尾延迟和 P95 尾延迟分别降低了 34%~77%, 26%~78%, 22%~78%; 在无丢包的条件下, 端到端平均延迟、P90 尾延迟和 P95 尾延迟分别降低了 31%~66%, 25%~69%, 25%~71%. 此外, 网程应用和全人工构建与管理的应用性能相当.

在未来工作中, 将采用更高效的编程语言(如 Golang, Rust)、先进的服务框架和基于共享内存的缓存机制优化网程系统, 提升其生命周期管理操作的速度. 此外, 将开发多种编程语言的虚拟运行时 API 和适配器, 以便接入更多的端边云运行时, 如 Serverless 运行时、大数据计算运行时, 进一步验证网程系统的通用性和性能. 为运行时调度器和运行时适配器设计高效的资源分配算法, 以减少运行时资源竞争.

作者贡献声明：俞子舒负责具体系统软件实现、实验分析和论文写作；王一帆负责实验环境搭建、部分实验案例设计和论文写作；曾琛负责部分实验结果分析和论文修改；张星洲负责部分实验案例设计；彭晓晖负责系统架构设计和论文修改；徐志伟负责系统理论抽象和架构设计.

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107−113

[2] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C/OL]//Proc of the 2nd USENIX Workshop on Hot Topics in Cloud Computing. Berkeley, CA: USENIX Association, 2010[2025-02-20]. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[3] Moritz P, Nishihara R, Wang S, et al. Ray: A distributed framework for emerging AI applications[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2018: 561−577

[4] Burns B, Grant B, Oppenheimer D, et al. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade[J]. Queue, 2016, 14(1): 70−93

[5] Yu Zishu, Li Fengzhi, Zheng Shoujian, et al. Can information superbahn achieve low entropy and high goodput? A validation study[J]. Chinese Journal of Computer, 2023, 46(11): 2302−2321 (in Chinese)
(俞子舒, 李奉治, 郑守建, 等. 信息高铁的低熵高通量性质验证[J]. 计算机学报, 2023, 46(11): 2302−2321)

[6] Shi Weisong, Zhang Xingzhou, Wang Yifan, et al. Edge computing: State-of-the-art and future directions[J]. Journal of Computer Research and Development, 2019, 56(1): 69−89 (in Chinese)
(施巍松, 张星洲, 王一帆, 等. 边缘计算: 现状与展望[J]. 计算机研究与发展, 2019, 56(1): 69−89)

[7] Xu Zhiwei, Chao Lu, Peng Xiaohui. T-REST: An open-enabled architectural style for the Internet of things[J]. IEEE Internet of Things Journal, 2018, 6(3): 4019−4034

[8] Xiong Ying, Sun Yulin, Xing Li, et al. Extend cloud to edge with KubeEdge[C]//Proc of the 3rd IEEE/ACM Symp on Edge Computing. Piscataway, NJ: IEEE, 2018: 373−377

[9] Chao Lu, Peng Xiaohui, Xu Zhiwei, et al. Ecosystem of things: Hardware, software, and architecture[J]. Proceedings of the IEEE, 2019, 107(8): 1563−1583

[10] Infiniband Trade Association. InfiniBand roadmap[EB/OL].[2024-06-10]. https://www.infinibandta.org/infiniband-roadmap/

[11] Chester J. Knative in Action[M]. New York: Simon and Schuster, 2021

[12] Oakes E, Yang L, Zhou D, et al. SOCK: Rapid task provisioning with serverless-optimized containers[C]//Proc of the USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 57−70

[13] Khan B S, Jangsher S, Ahmed A, et al. URLLC and eMBB in 5G industrial IoT: A survey[J]. IEEE Open Journal of the Communications Society, 2022, 3: 1134−1163

[14] Van Nee R. Breaking the gigabit-per-second barrier with 802.11ac[J]. IEEE Wireless Communications, 2011, 18(2): 4−4

[15] Sheshadri R K, Koutsonikolas D. On packet loss rates in modern 802.11 networks[C]//Proc of the IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2017: 1−9

[16] Robert W, Senior A. USA 5G experience report[EB/OL]. 2023[2024-06-10]. https://www.opensignal.com/reports/2023/01/usa/mobile-network-experience-1

[17] Ian F. Benchmarking the global 5G experience—June 2022[EB/OL]. 2022[2024-06-30]. https://www.opensignal.com/2022/06/22/benchmarking-the-global-5g-experience-june-2022

[18] Zhao Yong, Wei Mingshuo, Hu Chunlei, et al. Latency analysis and field trial for 5G NR[C]//Proc of the 17th IEEE Int Symp on Broadband Multimedia Systems and Broadcasting. Piscataway, NJ: IEEE, 2022: 1−5

[19] Al-Sarawi S, Anbar M, Alieyan K, et al. Internet of things (IoT) communication protocols[C]//Proc of the 8th Int Conf on Information Technology. Piscataway, NJ: IEEE, 2017: 685−690

[20] Dong Chao, Tao Ting, Feng Simeng, et al. Overview on medium access control protocol in flying ad-hoc NETworks and vehicular ad-hoc NETworks[J]. Journal of Electronics & Information Technology, 2022, 44(3): 790−802 (in Chinese)
(董超, 陶婷, 冯斯梦, 等. 面向无人机自组网和车联网的媒体接入控制协议研究综述[J]. 电子与信息学报, 2022, 44(3): 790−802)

[21] Xu Zhiwei, Li Guojie, Sun Ninghui. Information superbahn: Towards new type of cyberinfrastructure[J]. Bulletin of Chinese Academy of Sciences, 2022, 37(1): 46−52 (in Chinese)
(徐志伟, 李国杰, 孙凝晖. 一种新型信息基础设施: 高通量低熵算力网(信息高铁)[J]. 中国科学院院刊, 2022, 37(1): 46−52)

[22] Xu Zhiwei, Li Zhenying, Yu Zishu, et al. Information superbahn: Towards a planet-scale, low-entropy and high-goodput computing utility[J]. Journal of Computer Science and Technology, 2023, 38(1): 103−114

[23] Gatev R. Introducing Distributed Application Runtime (Dapr)[M]. Berkeley, CA: Apress, 2021

[24] Daniel J. Data Science with Python and Dask[M]. New York: Simon and Schuster, 2019

[25] Li Zijun, Guo Linsong, Chen Quan, et al. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing[C]//Proc of the USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2022: 69–84

[26] Jensen D. Beginning Azure IoT Edge Computing: Extending the Cloud to the Intelligent Edge[M]. Berkeley, CA: Apress, 2019

[27] Apache Software Foundation. Apache airflow[EB/OL].[2024-06-10]. https://airflow.apache.org/

[28] Zhuang Siyuan, Wang S, Liang E, et al. ExoFlow: A universal

workflow system for exactly-once DAGs[C]//Proc of the 17th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 269−286

[29] Parkhi O, Vedaldi A, Zisserman A. Deep face recognition[C/OL]// Proc of the British Machine Vision Conf 2015. Durham, UK: British Machine Vision Association, 2015[2025-02-20]. https://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf

[30] Grinberg M. Flask Web Development[M]. Sebastopol, CA: O'Reilly Media, 2018

[31] Ruan Zhenyuan, Park S J, Aguilera M K, et al. Nu: Achieving microsecond-scale resource fungibility with logical processes[C]// Proc of the 20th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 1409−1427

[32] Broadcom. VMware[EB/OL].[2024-07-18]. https://www.vmware.com/

[33] Proxmox Server Solutions GmbH. Proxmox virtual environment[EB/OL].[2024-07-18]. https://www.proxmox.com/en/

[34] The Centre for Speech Technology Research. The festival speech synthesis system[EB/OL].[2024-07-18]. https://www.cstr.ed.ac.uk/projects/festival/

[35] Baevski A, Zhou H, Mohamed A, et al. Wav2vec 2.0: A framework for self-supervised learning of speech representations[C]//Proc of the 34th Int Conf on Neural Information Processing Systems. New York: Curran Associates, 2020: 12449–12460

[36] Park M. IEEE 802.11 ac: Dynamic bandwidth channel access[C]// Proc of the 2011 IEEE Int Conf on Communications. Piscataway, NJ: IEEE, 2011: 1−5

[37] Zhang Kaipeng, Zhang Zhanpeng, Li Zhifeng, et al. Joint face detection and alignment using multitask cascaded convolutional networks[J]. IEEE Signal Processing Letters, 2016, 23(10): 1499−1503



Yu Zishu, born in 1996. PhD, assistant professor. Member of CCF. His main research interests include distributed systems and runtime management.

俞子舒, 1996 年生. 博士，助理研究员. CCF 会员. 主要研究方向为分布式系统、运行时管理.



Wang Yifan, born in 1992. PhD, associate professor. Member of CCF. His main research interests include distributed systems and resource management.

王一帆, 1992 年生. 博士，副研究员. CCF 会员. 主要研究方向为分布式系统、资源管理.



Zeng Chen, born in 1987. PhD, assistant professor. Member of CCF. Her main research interests include distributed ledge technology, access control, and edge computing.

曾　琛, 1987 年生. 博士，助理研究员. CCF 会员. 主要研究方向分布式账本技术、访问控制、边缘计算.



Zhang Xingzhou, born in 1992. PhD, associate professor, master supervisor. Member of CCF. His main research interests include distributed systems and programming methods.

张星洲, 1992 年生. 博士，副研究员，硕士生导师. CCF 会员. 主要研究方向为分布式系统、编程方法.



Peng Xiaohui, born in 1984. PhD, associate professor, PhD supervisor. Senior member of CCF. His main research interests include distributed systems and operating systems.

彭晓晖, 1984 年生. 博士，副研究员，博士生导师. CCF 高级会员. 主要研究方向为分布式系统、操作系统.



Xu Zhiwei, born in 1956. PhD, professor, PhD supervisor. Fellow of CCF. His main research interests include computer architecture and distributed systems.

徐志伟, 1956 年生. 博士，研究员，博士生导师. CCF 会士. 主要研究方向为计算机体系结构、分布式系统.

附录 A.

寻人应用使用的 UDP 协议如下：

数据包格式. 请求 ID(4 B)、数据包总数(4 B)、数据包序号(4 B)、数据部分(1 460 B).

客户端协议.

1)将待传输数据以 1 460 B 为单位分割成 N 个部分, 并为每部分数据创建一个数据包并赋予唯一序号.

2)发送 1 份完整数据至服务端.

3)接收服务端回复：若回复传输完成, 则退出; 否则, 记录未确认数据包的序号. 若超时, 则重传未确认的数据包. 超时时间设置为 1 ms. 重传和接收回复的过程在 2 个独立线程中进行.

服务端协议.

1)接收到数据包后进行解析, 记录接收的数据包的序号.

2)若全部数据包接收完毕, 则回复已完成; 否则, 回复未完成并附上收到的数据包的序号.