Accompanist：用于容错编舞式编程的运行时

VIKTOR STRATE KLØVEDAL，南丹麦大学，丹麦DAN PLYUKHIN，南丹麦大学，丹麦MARCO PERESSOTTI，南丹麦大学，丹麦FABRIZIO MONTESI，南丹麦大学，丹麦

arXiv:2603.20942v1 [cs.PL] 21 Mar 2026

在面向服务的架构中，服务通过以下两种方式之一进行协调：直接通过点对点通信，或间接通过称为编排器的中介。编排器通常更受欢迎，因为其本地状态是正在进行的工作流状态的“唯一真实来源”，这简化了采用“Saga”模式的分布式事务的故障恢复和回滚。但受硬件约束和安全策略影响，编排并非总是可行。没有中央编排器时，正确实现容错的 Saga 事务十分困难。

一个自然的想法是采用编舞式编程，这种范式将编排器的“全局视图”带入去中心化环境。遗憾的是，编舞式编程依赖于对网络可靠性和服务正常运行时间的严格假设，而这些假设往往并不成立。近期研究通过加入“故障感知”语言特性放宽了其中一些假设，但这些特性会使程序更加复杂。我们提出一种互补的方法：共同设计编程接口与可定制的运行时，后者能够通过重放计算来掩盖故障。我们的方法使程序保持简洁，无需修改编译器，并有助于在形式化证明中实现清晰的关注点分离。

我们介绍 Accompanist，这是一种适用于 Choral 编排式编程语言的弹性运行时。借助 Accompanist，程序员可以将去中心化的 Saga 事务实现为编舞程序，并将编译后的代码部署到与现有代码库中的服务并行运行的“边车”中。我们的关键假设是：编舞程序应当具有确定性，Saga 中的事务应当幂等，并且消息应写入持久化消息队列。基于这些假设，我们提出了一个形式化模型，并证明目标代码在构造上即正确：编舞式 Saga 无死锁，在有界重启下会终止；终止时，其参与方要么全部成功，要么都已调用补偿事务来撤销工作流。

补充关键词　词语和短语：编排、编舞式编程、工作流、Saga 事务

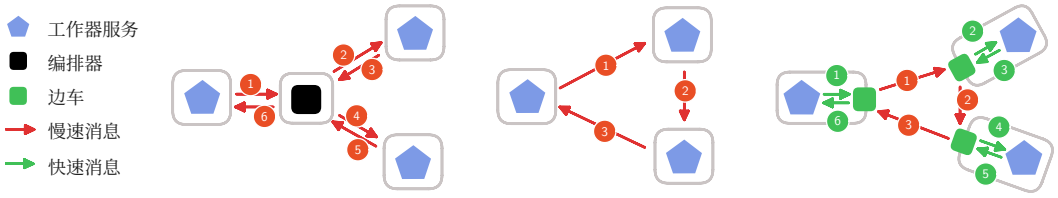
1 引言

将单个“单体”应用拆分为多个独立服务，原因有很多 [13, 38]。服务可以独立开发、部署和扩展。服务也可以用不同的编程语言编写，使组织能够利用更广泛的软件生态系统和人才库。通过让服务更简单、更小（微服务），程序员可以使应用更能抵御硬件和软件故障 [34]。然而，编写用于协调服务的工作流并非易事。

协调服务有两种方式：编排和编舞 [22]。在编排模式中，中央的“编排器”服务会向处理领域逻辑的“工作者”服务发起远程过程调用（RPC）（参见图1a）。许多组织指出，编排使复杂工作流更易于监控和维护 [12, 33]。其主要优势包括：

A1.（全局视图）整个工作流都在一个服务中运行，甚至可能都在一个过程中。A2.（一种语言）工作节点可以使用不同的语言，但编排器使用一种语言。A3.（安全性）无死锁性、确定性和消息类型安全性都很容易进行静态检查。

作者地址：Viktor Strate Kløvedal，viktorstrate@imada.sdu.dk，南丹麦大学，丹麦欧登塞；Dan Plyukhin，dplyukhin@imada.sdu.dk，南丹麦大学，丹麦欧登塞；MarcoPeressotti，peressotti@ imada.sdu.dk，南丹麦大学，丹麦欧登塞；Fabrizio Montesi，fmontesi@imada.sdu.dk，南丹麦大学，丹麦欧登塞。



(a) 编排 (b) 传统编舞 (c) Accompanist 编舞

图 1. 由三种不同风格实现的、包含三个服务的工作流。

A4. （独立部署）部署新的工作流时，无需重新部署工作节点。A5. （单一事实来源）发生故障时，可以通过查询编排器的本地数据库来恢复工作流。

然而，让所有交互都经过单个服务可能会造成不必要的网络跳转，每次跳转的耗时从微秒到数百毫秒不等 [24, 41]。不必要的跳转对端到端延迟造成的影响，我们称之为编排税。组织可以通过 RPC 链和分布式 future 等优化来尝试规避编排税 [44, 53]，也可以像 ServiceWeaver 那样动态合并编排器与工作服务 [16]。但有时这还不够：优化无法让程序员完全控制数据移动，而且有些服务因绑定到特定硬件而无法共置 [38]。安全策略和数据主权要求也可能对服务的协调方式施加严格限制，并彻底排除使用编排器的可能 [2, 35, 36, 41]。

没有编排时，工作器服务需要采用编舞：直接进行点对点通信，且没有固有的控制中心（参见图 1b）。传统上，编舞会将工作流的实现拆分到多个不同的服务中，从而牺牲了A1–A5所带来的优势。 编舞式编程 [30, 31] 有助于恢复其中一些优势。其核心思想是扩展传统语言，引入“角色”（值或计算所在的位置）和“通信”（将值从一个位置传送到另一个位置的方式）的概念。由此产生的源代码，我们称之为编排式程序，可以编译生成作为目标代码的编舞 。研究人员已开发出基于 Haskell [42], Rust [6],Elixir [56], Clojure [1], 等语言的编排式编程语言；本文将重点介绍 Choral [18], ，一种基于 Java 的编排式编程语言。

遗憾的是，现实中的面向服务的应用有一长串需求，任何编舞式编程语言都无法满足：

R1. （并发性）服务不应为了参与另一个工作流而必须先完成当前工作流。R2. （一致性）工作流结束时，所有参与方要么全部成功，要么全部失败。R3. （持久性）即使发生崩溃或消息丢失，工作流仍能取得进展。R4. （响应性）工作流启动后，服务可以在运行时添加新的参与方。

要求R1、R2和R3 是 *Saga* 事务 [15]的特性。Saga 是一种分布式事务，它牺牲了经典 ACID 事务所要求的一致性和隔离性，从而提升了面向服务的应用的性能。我们的核心挑战是设计一种架构，在缺少 R1–R4 这些优势的情况下，仍能满足这些要求并具备优势A1–A4，尽管缺少 A5。

研究人员一直致力于通过让编舞式编程更具表达力来满足各项要求R1–R4 。Graversen 等人提出了框架，以从丢失消息中恢复 [20], Plyukhin 等人提出了一种新的语义，以提高并发性 [37], Lugović 和 Montesi 提出了一种事件驱动模式，使编舞具有响应性 [27], ，而 Wiesdorf 和 Greenman 提出了同步检查点，以实现全有或全无语义 [56]。这些扩展各自只

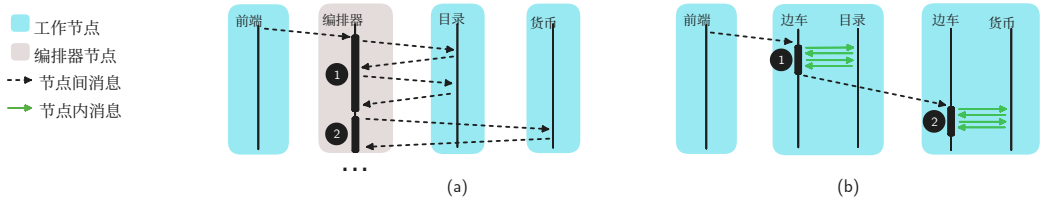


图 2. 使用编排（a）与 Accompanist（ b).

满足我们部分要求，而其中一些还需要对编译器进行非平凡的修改。一些语言扩展还会让程序变得更加复杂。将它们全部整合到一处，会削弱编排器与编舞程序之间优雅的对应关系。

我们的解决方案是Accompanist：一种用于容错编舞程序的语言运行时。程序员通过编写普通的 Choral 程序，并将编译后的代码部署到边车 [29, 39]（如图1c 所示）来使用边车，它们伴随工作器服务运行。边车按照 Choral 程序的指示进行点对点通信（图1c 中的三条红色箭头），并通过远程过程调用与工作器服务通信（图1c 中的六条绿色箭头）。与假设整个应用程序都以编舞式语言编写的近期研究不同 [6, 18, 27, 37, 42, 56], 我们的方法可以平滑地集成到现有的多语言应用中。事实上，从工作器服务的角度来看，来自边车的远程过程调用与来自编排器的远程过程调用并无不同；不过，我们预计前者的延迟更低，因为边车可以部署在与工作器服务相同的位置。

在默认模式下，Accompanist 提供监控、响应性、资源多路复用和至多一次语义等基本功能。我们通过移植两个开源微服务应用中的编排器，展示了默认模式的实用性；并观察到，当服务部署在不同可用区或区域时，端到端延迟有所改善。Accompanist 还提供容错模式，会重新运行编译后的 Choral 程序，直到程序发出完成或故障信号。我们的初步发现表明，在容错模式下运行编舞式 Saga 事务的开销并不高；事实上，由于架构差异，与业界使用的成熟 Saga 编排框架相比，我们观察到端到端延迟中位数改善了 5.9× 。

尽管语言运行时和基于重放的恢复并非新概念，但我们的工作首次将它们应用于编舞式编程，而评估结果表明，这种方法仍能实现良好的性能。我们以一项全新的理论成果补充这些实证结果：首次对去中心化的构造即正确 Saga 事务进行了形式化。我们的方法清晰地分离了以下关注点：（1）在编舞程序层面证明无死锁性和 Saga 安全性；（2）证明编舞程序与编译后的编舞之间满足标准的双模拟结果；（3）证明在网络层面，每个包含重启的执行都等价于无故障执行。

在接下来的章节中，我们将先简要介绍面向服务的架构、编舞式编程及其局限性。随后，我们会在第 Accom-panist 节概述其运行时4，并以此引出第 5 节形式化模型中的构造。最后，我们将在第 6 节进行性能评估，在第 7 节介绍相关工作，并在第 8 节展望未来工作。

2 动机：两个案例研究

2.1 Online Boutique

Online Boutique [19] 是谷歌开发的一款开源 Web 应用，用于展示现代微服务架构。该应用中的服务使用 C#、Java、Go、Python 和 JavaScript 编写，并通过 gRPC [21]进行通信。我们将 Online Boutique 代码库作为案例研究，以识别编排式微服务的一些最佳实践和常见问题。

2.1.1 编排器。处理领域逻辑的工作服务（例如商品目录或计费服务）以远程过程调用服务器的形式实现。这些服务器在本地处理远程过程调用，彼此之间不交互，从而最大限度地减少耦合。

大多数用户请求都由应用前端通过向工作服务发起少量远程过程调用来处理；例如，查看用户的购物车是通过向购物车服务发起远程过程调用实现的。对于涉及许多服务的复杂交互，相关功能会被拆分出来，由专门的编排器服务负责。例如，当用户发起结账请求时，前端会将其委托给一个名为结账服务的编排器。

与直接从前端发起远程过程调用相比，使用专门的编排器处理复杂工作流有几个优点：

(1) 编排中的编程错误不会影响前端中的其他 API 端点。(2) 运维人员可以将编排器部署在靠近其通信对象工作节点的位置。(3) 编排器和前端可以由不同的团队维护，并且可以独立扩展。

然而，编排的扩展性会受到网络延迟的严重影响。接下来我们将看到，为了缓解这种延迟，工作服务需要开发专用 API，供其客户端使用。

2.1.2 不必要的顺序执行。或许令人惊讶的是，Online Boutique 中的所有服务都使用阻塞式 RPC，而非非阻塞式 RPC。阻塞式 RPC 易于使用，因为它们与流行的命令式编程语言中的普通过程调用非常相似；相比之下，非阻塞式 RPC 的 API 更为复杂，可能会引入不必要的非确定性和分布式并发错误 [25]。尤其是 gRPC 库，并不保证非阻塞式调用会按照程序顺序处理。因此，开发人员为了提高可维护性而偏好阻塞式 RPC，也就不足为奇了。

2.1.3 非本地计算。结账服务不仅使用阻塞式 RPC，还会在循环中执行这些调用。如图2a所示，结账服务会：(1) 请求商品目录获取购物车中每件商品的价格，然后 (2) 请求货币服务将每件商品的价格转换为用户的本地货币。当然，结账服务若能通过一次远程过程调用传入用户购物车中的所有商品列表，会更简单、更快速；但关键在于，商品目录只提供获取单件商品价格的方法。

这个简单的例子揭示了一个根本问题：要获得良好的性能，就必须针对编排器对工作服务进行专门化，但工作服务的开发人员无法预知编排器可能以哪些方式使用这些服务。在 Online Boutique 中，结账团队完全可以合理地要求产品团队添加一种处理商品项列表的方法。但在大型组织中，产品团队可能需要很长时间才能响应；如果产品团队属于独立的合作伙伴组织，这项请求甚至可能永远得不到满足；即使请求得到及时处理，也可以预见，产品目录 API 会逐渐臃肿起来，加入各种面向特定客户端的专用方法。因此，性能方面的考量会导致编排器与工作服务之间产生*API* 耦合 。

(a) Java 中的编排

```
1  // class Checkout
2  订单结账(请求 请求) {
3      商品项 items = 购物车服务.getCart(req);
4      货币 total = items.总成本();
5      订单 order = 计费服务.收费(total);
6      返回 order;
7  }
```

(b) Choral 中的编舞

```
1  // class 结账@(前端,                    购物车, 计费)
2  GrpcChannel@(前端,          购物车) ch1;
3  GrpcChannel@(购物车,  计费) ch2;
4  GrpcChannel@(计费,      前端) ch3;
5  订单@前端 结账(请求@前端请求) {
6      请求@购物车请求C = ch1.com(请求);
7      商品项@购物车 商品 =购物车服务.getCart(请求C);
8      货币@购物车 总计 =商品项.总成本();
9      货币@计费总计B = ch2.com(总计);
10     订单@计费 订单 = 计费服务.收费(总计B);
11     return ch3.com(order);
12 }
```

(c) Java 编舞

```
1  // class 结账_前端
2  GrpcChannel_A通道1; GrpcChannel_B 通道3;
3  订单结账(请求req) {
4      ch1。com(req); // 发送
5      return (订单) ch3.com();  // 接收
6  }
7  // 类 结账          _Cart
8  GrpcChannel_B ch1; GrpcChannel_A ch2;
9  void 结账(){
10     请求 reqC =(请求) ch1.com(); // 接收
11     商品项 items =购物车服务.getCart(用户标识);
12     货币 总成本 =items.总成本();
13     ch2.com(总成本);// 发送
14 }
15 // 类 结账_计费
16 GrpcChannel_B ch2; GrpcChannel_A ch3;
17 void 结账(){
18     货币 totalB =(货币) ch2.com();// 接收
19     订单 order = 计费服务.收费(totalB);
20     第3章。com(订单);// 发送
21 }
```

图3. 一种简单但效率低下的编排方式（a）；一种更高效的 Choral 编舞版本（b）；以及等效的 Java编舞（c）。手动将图 3a 转换为图 3c 需要“切碎”代码，从而破坏程序结构和安全保证。

2.2 DeathStar 酒店预订

作为另一个案例研究，我们考察了*DeathStar* [14]：这是一套采用多种技术的多语言微服务应用。我们特别关注酒店预订 应用，因为它呈现了一个现实场景：多个组织协同合作，由预订服务向远程工作服务发出请求，而每个工作服务都管理着某种有限资源（例如酒店和航班），访问这些资源可能需要耗费大量资源的广域网往返。我们发现，酒店预订与 Online Boutique 具有相似的特点：对远程服务的查询由编排器通过阻塞式 RPC 管理；尽管搜索编排器不会在循环中执行 RPC，但它确实体现了 API 耦合：工作服务提供了专用端点，用于一次处理有关多家酒店的查询。

通过编排，程序员无需修改任何工作器服务，即可为微服务应用添加复杂的工作流。然而，在高延迟环境中，编排的集中式特性可能会带来显著开销。当编排无法扩展时，我们希望有一种替代方案，既能提供同等程度的模块化，又能更好地控制数据局部性。

3 CHORAL 与编舞式编程

本节将介绍编舞式编程，作为解决第 2 节所述问题的一种自然方案。作为贯穿全文的示例，我们将使用图 3a 中的简单 Java 编排，该示例基于 Online Boutique 案例研究。编排器首先向购物车服务发起远程过程调用，以获取用户的购物车（第 3 行）；然后计算商品项的总成本（第 4 行），向计费服务发起远程过程调用（第 5 行），并返回结果。

当网络延迟较高时，编排式方案效率低下：购物车服务将商品项发送给编排器，而编排器只是把价格加总后，再将总额转发给

计费服务。如果我们的编程语言能让购物车服务在本地计算成本，并将总成本直接发送给计费服务呢？这样的语言需要能够指定计算应该“在哪里”进行，以及数据应该“如何”移动。编舞式编程是一种简单的底层抽象，恰好提供了这一功能。

3.1 迁移到 Choral

Choral [18] 编译器为 Java 引入了编舞式编程。我们可以通过在类名后注释一个结[illegible]类的类名后添加一个3a 中的类转换为 Choral 编舞，只需在类名后添加一组位置（也称为角色）：结账@(A,B,C) 是[illegible]包含位置 A、B 和 C 的 Choral 类。位置名称可以任意指定，Choral 编译器会将带注释的类转换为三个名为 结账_A、结账_B 和 结账_C 的 Java 类：

```
// Choral 编舞
类 结账@(A,B,C) {
  ...
}
```

Choral 编译器 →

```
// Java编舞类结账_A
{...}类结账_B {...}类
结账_C {...}
```

在 Choral 类中，我们通过标注类型来指定每个数据项的位置。请求@Frontend req 是一个位[illegible]nten[illegible]译后的代码中，这[illegible]显示为 请求 req 位于 结账_Frontend，且不会出现[illegible]类中。对图 3a 中的简单编排器添加标注后，我们得到如下结果：

```
//class Checkout@(Frontend,Cart,Billing)订单
@Frontend checkout(Request@Frontend req)
{Items@Cart items = cartSvc.getCart(req);货币
@Cart total = items.totalCost();订单@Billing
order  = billingSvc.收费(total);return order;}
```

注意，我们指定了 total[illegible]型为 货币[illegible]这意味着购物车服务会在本地计算总成本。

尽管上述代码在语法上正确，但 Choral 的类型检查器会拒绝它：变量req 位于[illegible]ntend，但 cartSvc 位于[illegible]购物车。在 *X* 等高级分布式编程语言中10 [11]，，通信将由不透明的运行时层管理。Choral 编排是一种低层次的抽象，因为它们不会增加运行时开销，而且所有通信都通过调用用户定义对象的方法显式完成。

为了让代码通过类型检查，我们通过实现一对扩展 Java 接口的类来创建一个通道 DiChannel_A 和 DiChannel_B。对[illegible]nist,[illegible]100 行纯 Java 代码实现了一个 gRPC 通道。

```
类 GrpcChannel_A<T>
实现 DiChannel_A<T>{// 发送方
法<M扩展 T> Unit com(M m)
{...
}}
```

```
类 GrpcChannel_B<T>实现
DiChannel_B<T>{// 接收方法<
M 扩展T> M com() {...}}
```

Choral 可将 GrpcCha[illegible]hanne[illegible]类 GrpcChannel@(A,[illegible]供了一个 com 方法，通过调用我们自定[illegible]通道上的方法，将 M@A 类型[illegible]对象转换为 M@B 类型的[illegible]象。

使用我们的自定义通道，完整的 Choral 编排器版本如图 3b 所示。在第 6 行，前端向购物车服务发送请求。在第 7 行，购物车服务获取用户购物车。在第 8 和 9 行，购物车服务在本地计算总成本，并将结果发送给

计费服务。在第10 和 11 行，计费服务向用户的银行卡收费，并将结果发回前端。尽管 Choral 编舞比编排更冗长，但它使我们能够更准确地描述数据的位置及其移动方式。

Choral 编译器生成的 Java 代码如图3c所示。值得注意的是，除了我们在图3b中显式编写的代码之外，Choral 不会增加任何运行时开销。这并不是因为 Choral 编译器进行了复杂的优化；它执行的转换相当简单，而且根据我们的经验，很容易预测源代码与目标代码之间的对应关系。例如：

（1）图3b 第5 行的方法标头被转换为图3c、9和17 中的方法标头。图3c中，请求 参数和订单 返回类型都出现在结账_前端 [illegible]而不会出现在另[illegible]个类中。

（2）图3b第6行的通信被转换为图3c 第4 行的“发送”操作，以及图10 第3c行的“接收”操作，对应于我们手写的GrpcChannel类中的方法。

我们将在第 3.3 节讨论这种翻译的局限性。但首先，我们总结一下这种新编程模型的优势。

3.2 编舞式编程模型

图 3c 中的 Java 代码完全可以手动编写。Choral 作为前端更加便捷，主要有三个原因：

结构。 Choral 的编舞代码与编排结构相吻合：从图 3a 迁移到图 3b很直接，而从图 3a迁移到图3c 则不那么容易。注意，在图 3c中，当前端向购物车服务发送消息时，控制流会从第 10 行跳到第 4 行；从第 13 行跳到第 18 行，以及从第 20 行跳到第 5 行的情况也类似。相比之下，Choral 代码的控制流始终位于单个过程中，并遵循简单的直线模式。Choral 还支持条件控制流和递归，因此即使在复杂的工作流中，也能反映编排器的结构。

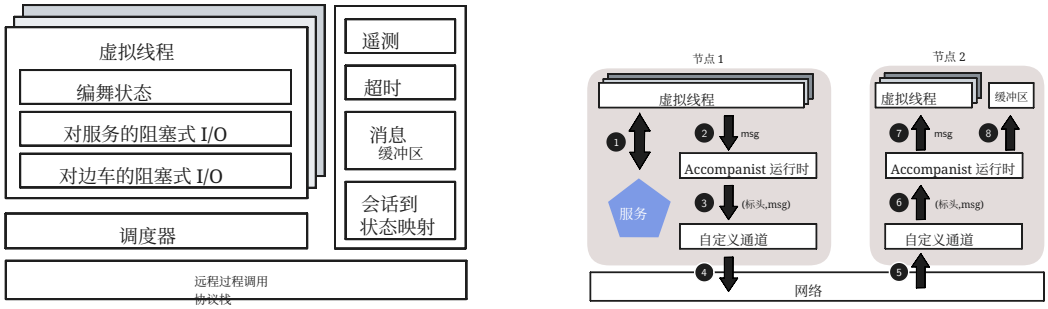
安全性。 如果程序员手动将图 3a转换为图 3c ，就很容易出错——例如忘记在第 20 行发送消息，或将第 4 行的消息强制转换为错误的类型。在我们的教学示例中，这类错误很容易发现；但当存在条件控制流和复杂的输入前置条件时，就没那么容易发现了 [25]。Choral 编译器可确保生成的代码无死锁、类型安全，且不存在通信不匹配 [18]。

确定性隐式并行。 编舞式编程提供了一种隐式并行的并发模型：只要满足以下条件，任意两个表达式就可以并行执行：(1) 它们位于不同的位置；(2) 两个表达式互不依赖 [31]。该并发模型也具有确定性，也就是说，只要服务不共享状态，所有执行都会产生相同的效果并返回相同的结果。因此，将编排器转换为 Choral 编舞可以保留原有行为，还可能引入原编排中没有的安全额外并行性。在某些情况下，非确定性并发模型能提供更高的并行性；程序员可以选择采用这种模型来处理这些情况 [37]。

3.3 局限性

编舞式编程（CP）是一种便捷的编程模型，但现有的编舞式编程方案存在显著局限，使其难以在实际系统中应用：

D1. CP re 要求服务阻塞，以便它们无法同时参与多个编舞 e;



(a) Accompanist 边车概览。 (b) 边车之间的通信。

图 4. Accompanist 边车

标头	描述
会话 *ID*	唯一标识编排调用。
编舞 *ID*	唯一标识实现该编舞的 Choral 类。
发送方 *ID*	唯一标识发送该消息的边车。
序列号	用于统计此发送方发送给此接收方的消息数量的序列号在此会话中。
遥测	用于将 OpenTelemetry 跟踪片段与其会话相关联的元数据。

表 1. 附加到消息的标头。

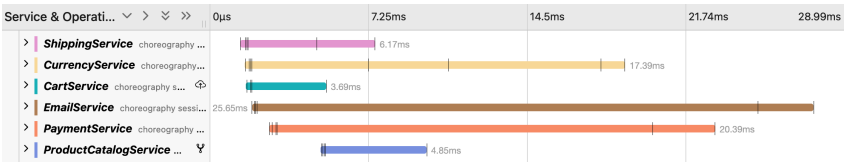


图 5. Accompanist Online Boutique 的遥测可视化。

D2. 编舞式编程假设网络可靠且服务不会发生故障；D3. 编舞式编程不允许服务加入正在运行的编舞；D4. 编舞式编程要求从头重写应用程序。

例如，在图 3的第 3c行，购物车服务预计会一直阻塞，直到接收到请求；因此，一位用户结账缓慢会对流水线中的其他用户产生连锁影响。这个阻塞调用还假定通信是可靠的；消息丢失或重复可能导致数据损坏 [37]。通信投影语言也不是响应式的：在图 3c 中，前端、购物车服务和计费服务都必须提前知晓，以便调用各自的 结账 方法。先前试图让通信投影语言更具响应性的做法，需要修改编舞：要么将其“切碎”为事件处理程序 [27]，要么牺牲确定性 [37]。

在下一节中，我们将展示如何在运行时层而非语言层解决这些限制，从而让开发者能够在真实系统中使用 Choral 的编程模型。

4 使用 ACCOMPANIST 将编排器去中心化

为了让 Choral 适用于面向服务的应用，我们开发了 Accompanist。该运行时支持在现有代码库中逐步以效率更高的编舞替换高延迟的编排器——而无需修改任何底层服务。

4.1 概述

图 4a 展示了运行时环境的概览。每次调用 Choral 编舞时，都会将其映射到一个轻量级虚拟线程；线程的栈封装了边车在编舞执行期间所需的所有本地状态；当线程向工作服务发起阻塞式 RPC，或等待来自另一个边车的消息时，它会被挂起。每次编舞调用都称为会话，运行时环境会维护会话 ID 与虚拟线程之间的映射。运行时还会为每个会话维护遥测信息、用于处理故障的超时机制，以及用于处理乱序消息的消息缓冲区。边车之间的通信由网络协议栈处理，程序员可通过 Choral 的通道 API（第3.1节）管理该协议栈。

Accompanist 通过为发出的消息添加标头来复用网络，相关内容概述于表 1。首次调用编舞时，发起方会生成一个唯一的会话 ID。编舞 ID 用于让服务动态加入正在运行的编舞，具体说明如下。发送方 ID 和序列号标头用于处理故障和乱序传递，相关内容也将在下文说明。遥测标头使 Choral 编舞能够与分布式追踪和日志工具集成，如图5所示。

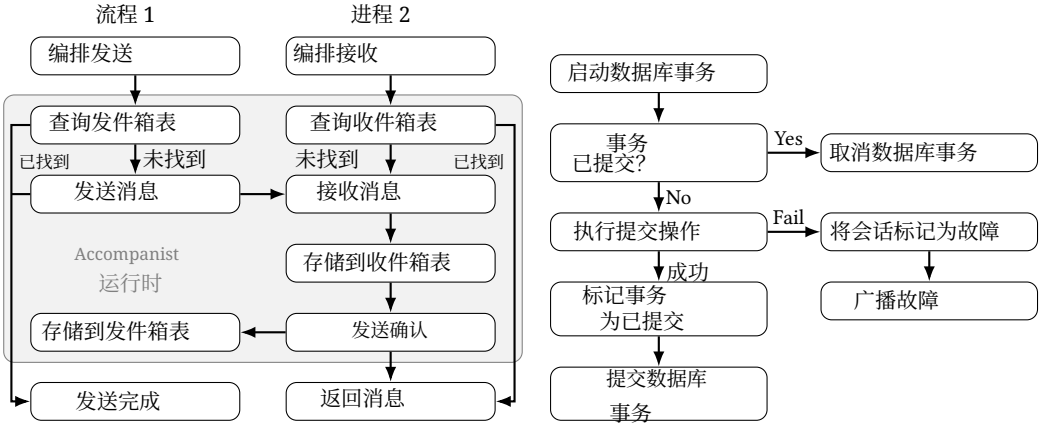
4.2 通信

图 4b 展示了我们框架中两个节点之间的编舞交互。在步骤 ●，Choral 生成的目标代码在虚拟线程中运行，并通过阻塞式 RPC 与关联的工作器服务交互。这些 RPC 效率很高，因为边车和工作服务始终部署在同一节点上，而且原线程阻塞时，其他虚拟线程仍可执行。值得注意的是，边车使用 RPC 而非进程间通信，因此无需修改工作器服务；这样一来，Accompanist 编舞便可与传统编排器共存。

在图● 的步骤2中，节点 1 上的边车向节点 2 上的边车发送一条消息。运行时环境会拦截该消息，并添加表 1所示的标头，然后将消息转发到用户定义的通信通道（步骤 ）。如果消息在步骤 – 到达目的地，接收方的运行时会检查消息标头。如果接收方此前未见过该会话 ID，则会根据编舞 ID 确定要运行的编舞，并实例化一个新的虚拟线程；如果接收方此前见过该会话 ID，且序列号和发送方 ID 与预期值匹配，则会将消息传递给虚拟线程（步骤 ）。如果序列号或发送方 ID 与预期值不匹配，则会将消息缓冲起来，因为消息可能是乱序送达的（步骤 ）。如果编舞在收到消息之前超时（第 4.3节），运行时会丢弃该消息。

4.3 终止

当虚拟线程完成编舞（或耗时过长仍未完成）时，会话便会终止，其资源也会被回收。之后，任何与已终止会话相关的消息都会被忽略。虚拟线程还会通过检查每次通信附带的序列号来处理丢失或乱序的消息；未按顺序到达的消息会先缓冲，直到可以按序处理，或会话被终止。这使我们能够实现至多一次执行模型：故障可能导致编舞无法完成，但资源始终会被回收。



（a）收件箱 / 发件箱 [39] 位于Accompanist。

(b) Accompanist 中局部事务的流程。

图 6. 具有容错能力的持久通信流程 (a) 和事务流程 (b)

4.4 容错

Accompanist 还提供了容错模式，采用至少一次语义，前提是编舞中的所有操作都是确定性的[1] 且幂等。该扩展使用进程本地数据库存储每次编排调用的元数据：会话 ID、编舞名称及其状态——状态可以是“已启动”、“已失败”或“已完成”。为简便起见，所有故障都通过重启会话来处理。要添加可自定义的重试策略很容易，例如，可以为图 7b 中第 8 行的 `com`方法添加一个可选参数。

边车重启时，系统会检查数据库，查看是否有需要恢复的会话。Accompanist 通过 从初始状态重新播放 会话来恢复它们。这需要一个持久化消息队列，我们通过收件箱/发件箱模式来实现，如图 [39] 所示。6a。当 Choral 程序发送或接收消息时，它会与 Accompanist 运行时（以灰色阴影标示）交互。当运行时接收到消息时，消息内容会存储在持久化的“收件箱”表中，随后系统会确认该消息并将其返回给 Choral 程序。当运行时发送消息时，会安装一个异步监听器以等待确认，同时立即将控制权返回给 Choral 程序。当重启后的进程需要接收消息时，运行时会先查询收件箱，查看该消息是否已接收。同样，运行时会丢弃发件箱中已存在的任何待发送消息。如果在指定的超时时间内没有收到消息或确认，会话就会重启。超时机制确保消息会不断重新发送，直至成功；消息标识符则确保重复消息会被忽略。

4.5 Saga 事务

作为我们框架的一项应用，我们展示了如何在编排式编程语言中实现*Saga* 事务 [15] 。为了说明这一问题，请考虑一个购买产品的工作流。首先，产品服务执行一个局部事务，预留该产品。接下来，支付服务执行一个局部事务来处理支付。任一事务都可能失败：产品可能缺货，用户也可能资金不足。如果第一个事务失败，我们不希望向用户的银行卡收费；如果第二个事务失败，我们希望

[1]给定相同的初始状态，每次执行都会在每个进程中产生完全相同的消息轨迹，其中消息内容在应用逻辑层面必须等价。

```
类产品事务 实现 事务 {

字符串事务名称() {
  返回   "更新产品库存";
}

布尔值 提交(整数会话标识符,     SQL事务 trans) {
  整数 库存 = trans.exec("SELECT stockFROMproduct");
  如果 (stock  == 0) {返回 false; }
  事务。执行("UPDATE 产品 SET 库存  =库存 - 1");
  返回 true;
}
无效 compensate(整数sessionID,
                                SQLTransaction trans) {
  事务执行("UPDATE 产品 SET 库存 =库存 + 1");
}}
```

(a) 在 Java 中实现局部事务。

```
1  类ProductWorkflow@(产品,          支付) {
2    void run(
3      订单@产品 订单      ,
4      故障会话上下文@产品 ctx_prd        ,
5      故障会话上下文@支付 订单_pmt,
6    ) {
7      ctx_prd.事务(
8        new 产品事务@产品(order));
9      订单@支付订单_pmt  =ch.com(order);
10     订单_支付。事务(
11       新的 支付事务@支付(订单));
12   }
13 }
```

(b) Choral 中 Saga 事务的示例，使用容错扩展 Accompanist。

图 7. Accompanist 中具有容错能力的事务。

回滚产品预留。Saga 是由多个局部事务组成的分布式事务，并提供以下保证：要么 (a) 每个局部事务最终都成功，要么 (b) 某个事务失败，并且每个局部事务都被回滚。

借助 Accompanist，程序员通过扩展 事务接口来实现本地事务（图7a）。本地事务可以组合成编舞式 Saga，如图 7b所示。当进程尝试提交事务时，它会遵循图6b所示的流程。首先，它会启动一个新的数据库事务。然后检查该事务是否已提交（例如，会话重启后）。如果已提交，则取消新事务并继续执行；这可确保事务具有幂等性。如果该事务尚未提交，进程便会执行提交操作。如果提交成功，则将事务标记为已完成。如果提交失败，则将整个会话标记为失败，并回滚该会话此前的所有事务。随后，进程会向其他参与方广播会话故障，确保它们中止编舞并回滚其事务。

5 形式化模型

现在，我们将运用前几节中的思想，为编舞式 Saga 事务构建一个正确性由构造保证的形式化模型，其中进程可能随时崩溃并重启，回到其初始状态。乍看之下，这项任务似乎艰巨无比！但通过清晰地分离关注点，我们只需三个简单部分便能证明端到端正确性结果：

（1）首先，我们扩展普通的编舞式编程模型，加入可能成功、失败或得到补偿的事务。在这一层面上，很容易证明事务无死锁，并且要么（a）所有进程都成功，要么（b）所有进程都调用补偿事务。（2）其次，我们引入网络模型，并建立一个教科书式的双模拟结果，将编舞程序与网络层的编译后编舞关联起来——前提是网络无故障，即进程绝不重启。（3）第三，我们设计了一种新颖的效率关系（$\lesssim$），将无故障执行与易发生故障的执行关联起来。这一关系依赖于持久化消息队列的存在，以及进程具有确定性的假设。

通过将各部分组合起来，关于编舞程序的性质便可延伸至易发生故障的网络层面。这里我们着重介绍高层次的概念；精确的形式化定义见补充材料。

$C ::= I; C \mid \mathbf{0} \mid \mathrm{p}.e \to \mathrm{q}.x \mid \mathrm{p} \rightsquigarrow \mathrm{q}.x \mid \mathrm{p} \to \mathrm{q}[\mathrm{l}] \mid \mathrm{p} \rightsquigarrow \mathrm{q}[\mathrm{l}] \mid \mathrm{p}.x := e \mid \text{若 } \mathrm{p}.e \text{ 则 } C_1 \text{ 否则 } C_2 \mid \mathrm{p}.x := t(e) \mid e := v \mid x \mid f(\vec{e})$

$$\frac{\Sigma(\mathrm{p}) \vdash e \downarrow v \qquad \mathrm{p} \in A}{\langle \mathrm{p}.x := e; C, \Sigma, K, S, T, A\rangle \xrightarrow{\tau@\mathrm{p}} \langle C, \Sigma[\mathrm{p}.x \mapsto v], K, S, T, A\rangle}\ \textsc{C-local}$$

$$\frac{\langle C, \Sigma, K, S, T, A\rangle \xrightarrow{\mu} \langle C', \Sigma', K', S', T', A'\rangle \qquad \mathrm{p} \neq \mathrm{pn}(\mu)}{\langle I; C, \Sigma, K, S, T, A\rangle \xrightarrow{\mu} \langle I; C', \Sigma', K', S', T', A'\rangle}\ \textsc{C-delay}$$

$$\frac{i = S(\mathrm{p}, \mathrm{q}, \textsc{send}) \qquad \Sigma(\mathrm{p}) \vdash e \downarrow v \qquad \mathrm{p} \in A}{\langle \mathrm{p}.e \to \mathrm{q}.x; C, \Sigma, K, S, T, A\rangle \xrightarrow{\mathrm{p}.v \to \mathrm{q}!} \langle \mathrm{p}.e \rightsquigarrow \mathrm{q}.x; C, \Sigma, K[\mathrm{p}, \mathrm{q}, i \mapsto v], \mathit{incSend}(S, \mathrm{p}, \mathrm{q}), T, A\rangle}\ \textsc{C-send-val}$$

$$\frac{i = S(\mathrm{p}, \mathrm{q}, \textsc{receive}) \qquad K(\mathrm{p}, \mathrm{q}, i) = v \qquad v \neq \bot \qquad \mathrm{q} \in A}{\langle \mathrm{p} \rightsquigarrow \mathrm{q}.x; C, \Sigma, K, S, T, A\rangle \xrightarrow{\mathrm{p}.v \to \mathrm{q}?} \langle C, \Sigma[\mathrm{q}.x \mapsto v], K, \mathit{incRecv}(S, \mathrm{q}, \mathrm{p}), T, A\rangle}\ \textsc{C-recv-val}$$

$$\frac{\Sigma(\mathrm{p}) \vdash e \downarrow v \qquad t(v) = v' \qquad \mathrm{p} \in A}{\langle \mathrm{p}.x := t(e); C, \Sigma, K, S, T, A\rangle \xrightarrow{\tau@\mathrm{p}} \langle C, \Sigma[\mathrm{p}.x \mapsto v'], K, S, T[\mathrm{p} \Rightarrow t(v) = v'], A\rangle}\ \textsc{C-success}$$

$$\frac{\mathrm{p} \in A \qquad t \notin T(\mathrm{p})}{\langle \mathrm{p}.x := t(e); C, \Sigma, K, S, T, A\rangle \xrightarrow{\textsc{compensate}@\mathrm{p}} \langle C, \Sigma, K, S, \mathit{comp}(T, \mathrm{p}), A \setminus \{\mathrm{p}\}\rangle}\ \textsc{C-fail}$$

$$\frac{A \neq A_{start} \qquad \mathrm{p} \in A}{\langle C, \Sigma, K, S, T, A\rangle \xrightarrow{\textsc{compensate}@\mathrm{p}} \langle C, \Sigma, K, S, \mathit{comp}(T, \mathrm{p}), A \setminus \{\mathrm{p}\}\rangle}\ \textsc{C-compensate}$$

图 8. 编舞的部分语法和操作语义。

5.1 定义

网络中参与的进程名称集合记作 $\mathbb{P}$。消息状态，K: $\mathbb{P}\times\mathbb{P}\times\mathbb{N} \to$**值** $+ \bot$，是一个将发送方、接收方和序列号映射到某个值的函数；若不存在值，则映射到 $\bot$。

序列号状态，S: $\mathbb{P}\times\mathbb{P}\times\{\text{send, receive}\} \to \mathbb{N}$，是一个将本地进程、远程进程以及发送和接收这两个标签映射到序列号的函数。它用于跟踪进程之间下一次要发送和接收的序列号。

操作 *incSend*$(S, \mathrm{p}, \mathrm{q})$ 会生成一个序列号状态，其中 p 向 q 发送的计数已递增。同理，操作 *incRecv*$(S, \mathrm{p}, \mathrm{q})$ 会生成一个序列号状态，其中 p 从 q 接收的计数已递增。操作 *restart*(S, p) 会生成一个序列号状态，其中 p 的发送和接收计数均重置为零。

编舞存储 Σ： $\mathbb{P} \times$变量 $\to$ 值，表示编舞或网络中每个进程的本地状态。

一个事务状态, T: $\mathbb{P} \to$ [**Trans**], 是从进程名称到事务序列的映射，其中事务**Trans** 可以是提交操作 $t(v) = v'$，其含义是“提交事务 t并以 v为输入会产生结果 v'”，也可以是补偿操作 $c(t(v) = v')$。

5.2 编舞模型

我们假设语法中有一个由 x 表示的无界变量集合，以及进程名称 p、q、r。编舞 C要么为空 0，要么是序列 $I;C$，其中 I是一条指令， C是其后续程序。指令可以是以下任意一种。指令 p.e → q.x 表示 p 将一个值放入发给 q 的消息队列，而 p ⇝ q.x 表示 q 从消息队列接收消息。指令 p ⇝ q.x 仅作为运行时项存在，不允许直接用于编舞程序。指令 p → q[l] 和 p ⇝ q[l] 除了用于确定选择知识的通信标签外，其余相同。p.x := e 是局部赋值，而“if p.e then C_1 else C_2”表示在 p 处求值的条件语句。最后，事务指令 p.x := t(e)的含义是“进程 p 执行事务 t，以计算 e所得结果作为输入，并将事务结果存储在变量 x中”。容错编舞的语法总结于图8顶部。

编舞的配置是一个 6 元组 C, Σ, K, S, T, A，其中 A是活动进程（即未因故障而得到补偿的进程）的集合。初始配置是元组 $C_{start}, \Sigma_{start}, K_{start}, S_{start}, T_{start}, A_{start}$。

图 8 还展示了编舞操作语义中的部分选定规则。 C-LOCAL规则通过使用 e 和 p的本地状态来计算表达式，从而执行局部赋值。 C-DELAY规则刻画了编舞程序的乱序语义；它允许进程 q 在不等待其他进程执行 C的情况下，执行 I。

C-send-val 和 C-recv-val 规则定义了进程如何通信。发送消息时，会将其以原子方式追加到队列 K。实际上， K 可能会在远程服务上实现为持久化消息队列；在这种情况下，发送进程会反复尝试将消息加入队列，直到队列返回确认。我们可以安全地将此建模为原子操作，因为我们假设发送进程在收到确认之前不会继续执行。为建模进程接收消息的过程，C-recv-val 规则会从 K 读取一个值，并将其存储在进程 p 的本地状态中。序列号 S (q, p, receive) 用于对队列中的消息进行索引。

Saga 的语义由 C-success、C-fail 和 C-compensate 规则刻画。C-success 规则会将事务提交到事务存储中。在 C-fail 规则中，事务失败，p 会补偿其先前已提交的事务。一旦某个进程失败，其他所有进程最终也会使用 C-compensate 规则来补偿各自的事务。

在编排层，很容易看出，编舞要么规约为 **0** ，要么都执行补偿事务。以下两个定理对此进行了形式化表述。

引理 3 （编排的无死锁性）。若 s 是一个已终止的编排配置，则以下情况之一成立：

(1) s 具有形式 $0, \Sigma, K, S, T, A$，其中 Σ, K, S, T, A 为任意取值；或者 (2) s 具有形式 $C, \Sigma, K, S, T, \emptyset$，其中 C, Σ, K, S, T 为任意取值。

定理 2 （编排的弱原子性）。设 σ 为一次已终止的编排执行，s_{final} 为最终配置。那么以下情况必有其一：
(1) s_{final}的形式为 $0, \Sigma, K, S, T, \mathbb{P}$，其中 T没有补偿事务；或 (2) s_{final}的形式为 $C, \Sigma, K, S, T, \emptyset$，其中 T(p) 的形式为 $t_1, t_2, \ldots, t_i, c(t_i) \ldots, c(t_2)\ c(t_1)$，对于某个 i，对每个进程 p ∈ $\mathbb{P}$。

接下来，我们将解释编舞与进程可能发生故障的网络之间的关系。

$$P, Q, R ::= I; P \mid \mathbf{0}$$
$$I ::= \mathrm{p}!e \mid \mathrm{p}?x \mid \mathrm{p} \oplus \mathrm{l} \mid \mathrm{p}\&\{(\mathrm{l})_i : P_i\}_{i \in I} \mid x := e \mid \text{if } e \text{ then } P \text{ else } Q \mid x := t(e)$$
$$e ::= v \mid x \mid f(\vec{e})$$

$$\frac{\Sigma(\mathrm{p}) \vdash e \downarrow v \qquad \mathrm{p} \in A}{\langle \mathrm{p}[x := e; P], \Sigma, K, S, T, A\rangle \xrightarrow{\tau@\mathrm{p}} \langle \mathrm{p}[P], \Sigma[\mathrm{p}.x \mapsto v], K, S, T, A\rangle}\ \textsc{P-local}$$

$$\frac{\langle N, \Sigma, K, S, T, A\rangle \xrightarrow{\mu} \langle N', \Sigma', K', S', T', A'\rangle}{\langle N \mid M, \Sigma, K, S, T, A\rangle \xrightarrow{\mu} \langle N' \mid M, \Sigma', K', S', T', A'\rangle}\ \textsc{P-par}$$

$$\frac{i = S(\mathrm{p}, \mathrm{q}, \textsc{send}) \qquad \Sigma(\mathrm{p}) \vdash e \downarrow v \qquad \mathrm{p} \in A}{\langle \mathrm{p}[\mathrm{q}!e; P], \Sigma, K, S, T, A\rangle \xrightarrow{\mathrm{p}.v \to \mathrm{q}!} \langle \mathrm{p}[P], \Sigma, K[\mathrm{p}, \mathrm{q}, i \mapsto v], \mathit{incSend}(S, \mathrm{p}, \mathrm{q}), T, A\rangle}\ \textsc{P-send-val}$$

$$\frac{i = S(\mathrm{p}, \mathrm{q}, \textsc{receive}) \qquad K(\mathrm{p}, \mathrm{q}, i) = v \qquad v \neq \bot \qquad \mathrm{q} \in A}{\langle \mathrm{q}[\mathrm{p}?x; Q], \Sigma, K, S, T, A\rangle \xrightarrow{\mathrm{p}.v \to \mathrm{q}?} \langle \mathrm{q}[Q], \Sigma[\mathrm{q}.x \mapsto v], K, \mathit{incRecv}(S, \mathrm{p}, \mathrm{q}), T, A\rangle}\ \textsc{P-recv-val}$$

$$\frac{}{\langle \mathrm{p}[P], \Sigma, K, S, T, A\rangle \xrightarrow{\textsc{restart}@\mathrm{p}} \langle \mathrm{p}[N_{start}(\mathrm{p})], \Sigma(\mathrm{p} \mapsto \Sigma_{start}(\mathrm{p})), K, \mathit{restart}(S, \mathrm{p}), T, A\rangle}\ \textsc{P-restart}$$

$$\frac{\Sigma \vdash e \downarrow v \qquad t(v) \downarrow v' \qquad \mathrm{p} \in A}{\langle \mathrm{p}[x := t(e); P], \Sigma, K, S, T, A\rangle \xrightarrow{\tau@\mathrm{p}} \langle \mathrm{p}[P], \Sigma[\mathrm{p}.x \mapsto v'], K, S, T[\mathrm{p} \Rrightarrow t(v)], A\rangle}\ \textsc{P-success}$$

$$\frac{\mathrm{p} \in A}{\langle \mathrm{p}[x := t(e); P], \Sigma, K, S, T, A\rangle \xrightarrow{\textsc{compensate}@\mathrm{p}} \langle \mathrm{p}[P], \Sigma, K, S, \mathit{comp}(T, \mathrm{p}), A \setminus \{\mathrm{p}\}\rangle}\ \textsc{P-fail}$$

$$\frac{A \subset A_{start} \qquad \mathrm{p} \in A}{\langle \mathrm{p}[P], \Sigma, K, S, T, A\rangle \xrightarrow{\textsc{compensate}@\mathrm{p}} \langle \mathrm{p}[P], \Sigma, K, S, \mathit{comp}(T, \mathrm{p}), A \setminus \{\mathrm{p}\}\rangle}\ \textsc{P-compensate}$$

图 9. 网络的部分语法和操作语义。

5.3 网络模型

进程 P要么为空（0），要么是序列 $I; P$，其中 I是一条指令，P是其后续程序。指令可以是以下形式之一。指令 $\mathrm{p}!e$ 表示将表达式e 的结果发送到以 p 为接收方的消息队列中。类似地，$\mathrm{p}?x$ 表示从消息队列接收一个值（发送方为 p），并将其存入局部变量 x。$\mathrm{p} \oplus \mathrm{l}$ 表示向 p 发送一个标签；$\mathrm{p}\&\{(\mathrm{l})_i\colon P_i\}_{i \in I}$表示从 p 接收一个标签 l_i，然后继续执行与之关联的程序 P_i。$x := e$是局部赋值，而‘if $e$then Pelse Q’是条件语句。最后，$x := t(e)$ 指令在进程处执行一个局部事务。表达式 v、x 和 $f(\vec{e})$ 分别表示值、变量和函数调用。容错进程的语法总结于图9上方。

网络的配置是一个 6 元组 N，Σ，K，S，T，A，其中 N是一个进程网络。初始配置是一个元组 N_{start}，Σ_{start}，K_{start}，S_{start}，T_{start}，A_{start}。

图 9 还展示了进程操作语义的部分规则。P-local、P-send-val、P-recv-val、P-success、P-fail和P-compensate是图8中相应编排规则的进程版本。P-par 规则允许网络中的不同进程并发执行步骤。特殊的重启规则允许各个进程随时重新开始执行。消息状态 K 会在重启期间持续保留。

$$[\![\mathrm{p}.e \to \mathrm{q}.x; C]\!]_\mathrm{r} \triangleq \begin{cases} \mathrm{q}!e; [\![C]\!]_\mathrm{r} & \text{if r = p} \\ \mathrm{p}?e; [\![C]\!]_\mathrm{r} & \text{if r = q} \\ [\![C]\!]_\mathrm{r} & \text{otherwise} \end{cases} \qquad [\![\mathrm{p}.x := e; C]\!]_\mathrm{r} \triangleq \begin{cases} \mathrm{p}.x := e; [\![C]\!]_\mathrm{r} & \text{if r = q} \\ [\![C]\!]_\mathrm{r} & \text{otherwise} \end{cases}$$

$$[\![\mathrm{p} \rightsquigarrow \mathrm{q}.x; C]\!]_\mathrm{r} \triangleq \begin{cases} \mathrm{p}?x; [\![C]\!]_\mathrm{r} & \text{if r = q} \\ [\![C]\!]_\mathrm{r} & \text{otherwise} \end{cases} \qquad [\![\mathrm{p}.x := t(e); C]\!]_r \triangleq \begin{cases} x := t(e); [\![C]\!]_\mathrm{r} & \text{if r = q} \\ [\![C]\!]_\mathrm{r} & \text{otherwise} \end{cases}$$

图 10. 端点投影（EPP）的部分规则。

网络的终止定理指出，只要重启事件的数量不受限制，每个网络最终都会终止；否则，若某个进程不断无限重启，网络将永远无法取得进展。

定理 1（网络终止）。设 σ 为一次有限网络执行。那么存在一个终止执行 σ'，它扩展了 σ。此外，对于任意 $k \in \mathbb{N}$，都存在 $n \in \mathbb{N}$，使得：任何扩展 σ 且至多包含 k 个 P 重启事件的σ'，必须在至多 n 步内终止。反过来，如果 σ 是一次无限的已终止网络执行，那么 σ 有一个有限的已终止前缀。

请注意，并不能保证所有网络都无死锁；网络 p[q?x] | q[p?x] 已经终止，但我们称其为死锁的，因为进程并非全都**0**。然而，我们将看到，所有从编舞投影得到的 网络都是无死锁的。

图 10 展示了从编舞到基础网络中进程的端点投影函数。编舞发送指令 $[\![\mathrm{p}.e \to \mathrm{q}.x; C]\!]$ 投影后，在 p 处是发送操作，在 q处是接收操作。然而，编舞接收指令 $[\![\mathrm{p} \rightsquigarrow \mathrm{q}.x; C]\!]$投影后，仅在 q处是接收操作，因为此时来自 p 的消息已经发送。

编舞及其投影之间的关系由以下两个引理形式化，这两个引理共同定义了 EPP 定理：

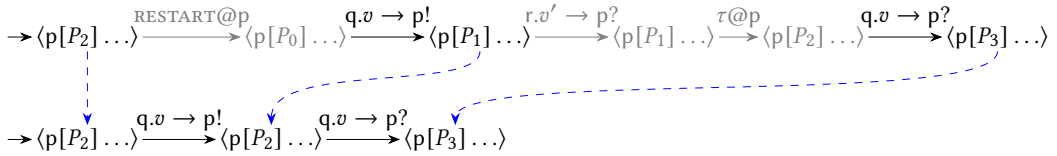
引理 1（端点投影的完备性）。对于所有 C, Σ, K, S, T, A、μ 和 $C', \Sigma', K', S', T' A'$，若 $C, \Sigma, K, S, T, A \xrightarrow{\mu} C', \Sigma', K', S', T' A'$，则 $[\![C]\!], \Sigma, K, S, T, A \xrightarrow{\mu} [\![C']\!], \Sigma', K', S', T' A'$。

引理 2（端点投影的可靠性）。对于任意 C, Σ, K, S, T, A $N, \mu, N' \Sigma', K'$ 和 S'，若 C, Σ, K, S, T, A 格式良好，$N \sqsupseteq [\![C]\!]$，且对于任意 p，$\mu \neq \mathsf{restart@p}$，则 $N, \Sigma, K, S, T, A \xrightarrow{\mu} N' \Sigma', K', S', T' A'$ 蕴含 $C, \Sigma, K, S, T, A \xrightarrow{\mu} C', \Sigma', K', S', T' A'$，其中存在某个 C' 使得 $N' \sqsupseteq [\![C']\!]$。

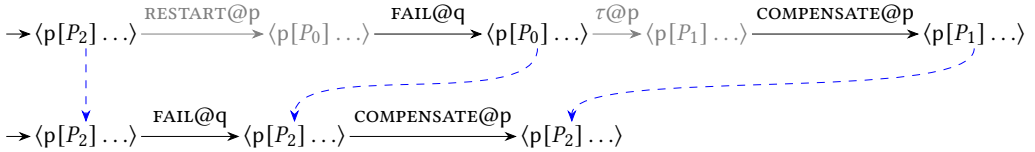
可靠性方向明显未涉及重启规则；当进程重启时，编排层究竟应该如何处理并不明确。相反，我们证明，带有重启的执行与不带重启的执行在形式上相关；接下来，我们将通过效率序（$\preceq$）对此进行形式化。

5.4 恢复

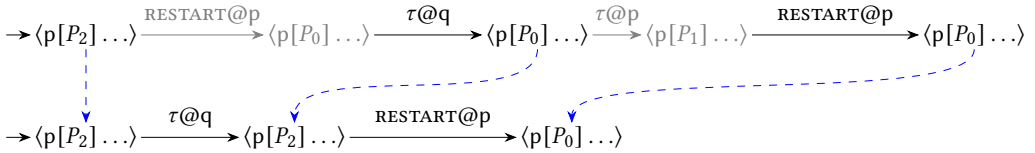
接下来，我们将展示进程语义如何使故障对编舞语义透明。具体而言，我们将证明，任何终止执行都与某个无重启执行“相关”。这是因为，当处于状态 P 的进程 p 重启时，它会重放从 P_{start} 开始的所有步骤，以恢复到状态P；这些“重放”步骤不会影响配置的其余部分，因为（1）事务是



（a）包含重启的网络执行（上图），以及与之相关的无重启执行（下图）。我们通过从上方的执行中剪去重启转换，以及将 p 恢复到其旧状态 P_2所需的任何其他转换，得到下方的执行。



(b) 带有重启的网络执行（上），以及与之相关的无重启执行（下）。在此示例中，进程 p 永远不会回到其旧状态 P_2，因为它会先执行补偿步骤。



(c) 带有重启的网络执行（上），以及少一次重启事件的相关执行（下）。在此示例中，进程 p 会在能够回到状态 P_2之前重启。

图 11

$$\frac{N'\restriction_A = N\restriction_A \qquad \Sigma'\restriction_A = \Sigma\restriction_A \qquad S'\restriction_A = S\restriction_A}{\langle N', \Sigma', K, S', T, A\rangle \cong \langle N, \Sigma, K, S, T, A\rangle}\ \text{CONG-1} \qquad \frac{s' \cong s \qquad \sigma' \cong \sigma}{s' \xrightarrow{\mu} \sigma' \cong s \xrightarrow{\mu} \sigma}\ \text{CONG-2}$$

$$\frac{\mathrm{p} = \mathrm{pn}(\mu)}{s_1 \preceq_{\mathrm{p}} s_1 \xrightarrow{\mu} s_2}\ \text{PREC-1} \qquad \frac{\mathrm{p} \neq \mathrm{pn}(\mu)}{s_1' \xrightarrow{\mu} s_2' \preceq_{\mathrm{p}} s_1 \xrightarrow{\mu} s_2}\ \text{PREC-2} \qquad \frac{\sigma_1' \preceq_{\mathrm{p}} \sigma_1 \qquad \sigma_2' \preceq_{\mathrm{p}} \sigma_2}{\sigma_1'\sigma_2' \preceq_{\mathrm{p}} \sigma_1\sigma_2}\ \text{PREC-3}$$

$$\frac{\sigma_1' = \sigma_1 \qquad \sigma_2' \preceq_{\mathrm{p}} \sigma_2 \qquad \sigma_3' \cong \sigma_3}{\sigma_1'\sigma_2'\sigma_3' \preceq \sigma_1\sigma_2\sigma_3}\ \text{PREC} \qquad \frac{\sigma_1 \preceq \sigma_2 \qquad \sigma_2 \preceq \sigma_3}{\sigma_1 \preceq \sigma_3}\ \text{PREC-TRANS}$$

图 12. 执行上的效率序关系定义。

是幂等的，并且（2）消息队列是持久的。此外，由于进程具有确定性，p 要么（1）回到状态 P，要么（2）失败并执行补偿事务。

这种行为实质上意味着，我们可以像重启从未发生过一样，对包含重启的执行进行推理，因为我们知道这些执行的结果是等价的。如11a图所示，如果忽略灰色标记的重启和重放步骤，两次执行便会变得等价。

图 11b 展示了一个更有趣的例子：在易故障执行中，p 会重启并进行补偿，然后才能到达其重启前的状态。尽管如此，我们仍可以认为顶部和

底部配置是“同余”的，因为它们的差异与我们的目的无关。图 12 给出的推理规则形式化了网络配置 $s \cong s'$ 与执行 $\sigma \cong \sigma'$之间的同余概念。规则 CONG-1 规定，如果从所有活动进程的角度来看，两个网络配置中的配置都相同，那么这两个网络配置就是同余的。根据规则 CONG-2，如果两个执行中的所有配置都分别同余，那么这两个执行就是同余的。

为定义（$\preceq$）关系，我们使用记号 $\sigma\sigma'$ 表示执行σ和 σ'的串接。只有当 σ 的结束配置与 σ'的起始配置相同时，才能进行串接。接着，如果对于 σ 或 σ'中的所有步骤 μ， μ 要么由 σ 和 σ' 共同执行（规则 prec），要么当 μ 由 p 执行时，它只在 σ' 中执行（规则 prec-1），要么当 μ 由另一个进程执行时，它只在 σ 中执行（规则 prec-2），我们就称从 p 的视角来看，执行 σ 先于 σ' （记作$\sigma \preceq_{\mathrm{p}} \sigma'$）。

这使我们能够陈述进程的恢复定理，并证明进程能够从重启中安全地恢复。

定理 3（恢复）。设 σ 是一个有限的网络执行，并在最后一步终止。如果 σ 有$k > 0$ 个重启事件，那么存在一个执行 σ'，它有 $k - 1$ 个重启事件，且 $\sigma' \preceq \sigma$。

结合恢复定理、EPP 定理和我们的编舞安全定理，最终可以得到网络的安全定理。

定理 4（网络安全）。设 σ 是一次已终止的网络执行。那么以下情况之一成立：（1）$\mathrm{cfg}(\sigma)$的形式为$\prod_{\mathrm{p}\in\mathbb{P}} \mathrm{p}[0]$， Σ， K， S， T， $\mathbb{P}$，其中 T不包含补偿；或者（2）$\mathrm{cfg}(\sigma)$的形式为N， Σ， K， S， T， $\emptyset$，其中 $T(\mathrm{p})$的形式为t_1， . . .， t_i， $c(t_i)$ . . .， $c(t_1)$，对于某个 i，对每个进程 $\mathrm{p} \in \mathbb{P}$均成立。

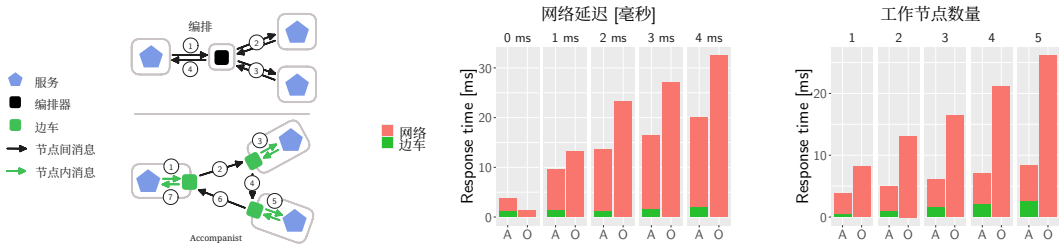
6 评估

到目前为止，我们已经论证，通过在工作服务旁部署边车，Accompanist 编舞能够实现比编排器更低的端到端延迟。本节将围绕以下研究问题评估我们的方法：

(1) 随着网络延迟增加，Accompanist的边车运行时开销与编排的开销相比如何？（第6.1节）(2) 在单个组织和单个数据中心内，Accompanist 编舞与编排器相比如何？（第6.2节）(3) 在不同组织和数据中心之间，Accompanist 编舞与编排器相比如何？（第6.3节）(4) 在端到端延迟方面，Accompanist Saga 与编排式 Saga 相比如何？（第6.4节）

6.1 微基准测试

为了深入理解 Accompanist 与编排的对比，我们将图3 中的简单工作流改编为微基准测试。图 3a 和图 3b 中的通信模式分别显示在图 13a的上方和下方。该图展示了 Accompanist 如何将编排服务分散到各个工作器服务旁运行的边车中。尽管边车会带来开销，因为服务之间无法直接通信，但我们预计，当网络延迟较高时，我们的框架仍能表现良好，因为 *Choral* 可以精确控制开销最高的通信。



(a) 通信图 (b) 两个等距工作节点 (c) n 工作节点 隔离，1 毫秒延迟

图 13. 微基准测试评估结果。

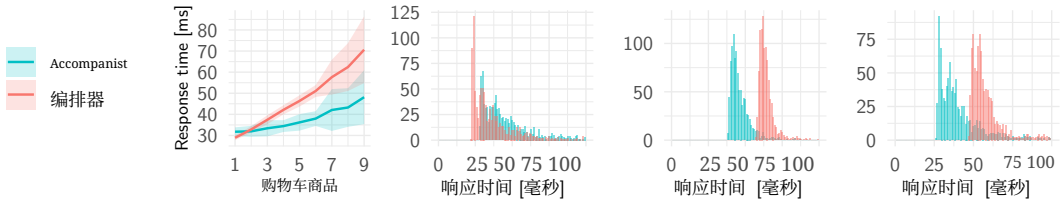
假设节点内消息耗时 t_1，节点间消息耗时 t_2，且计算时间可以忽略不计，图 13a 表明编排式方案的端到端延迟应为 $6t_2$，而 Accompanist 方案的端到端延迟应为 $6t_1+3t_2$；因此，当 t_1 保持不变而 t_2 增加时，我们的框架应比编排方案更具扩展性。我们通过在单台机器上运行微基准测试，并使用 Toxiproxy [43] 模拟节点间延迟，来评估这一预测。对于编排式基准测试，图 13b 展示了端到端延迟随网络延迟的变化。对于 Accompanist 基准测试，我们同时测量了端到端延迟以及服务与边车之间的延迟；通过用前者减去后者，我们估算出基准测试在执行开销较高的节点间通信和开销较低的边车 RPC 上分别花费了多少时间。结果证实了我们的预期：当网络延迟足够高时，Accompanist 比编排方案更具扩展性，因为无论网络速度如何，边车通信的开销都始终很小。

实际上，服务之间的网络延迟并不相同。例如，图 13a 中的工作服务可能位于与编排器不同的可用区或数据中心。在这种情况下，编排与编舞之间的性能差距会随着工作服务数量的增加而扩大。设 t_1 为节点内消息的延迟，设 t_2 为同一可用区内节点之间消息的延迟，并设 t_3 为跨可用区消息的延迟。那么，我们预计编排式方案的端到端延迟为 $2t_2 + 2nt_3$，而 Accompanist 的端到端延迟为 $2(n+1)t_1 + (n-1)t_2 + 2t_2$。这正是我们在图 13c 中观察到的现象：即使编排器与工作节点之间的延迟仅为 1 毫秒，编排开销仍会随着服务数量的增加而增大，因为工作节点之间进行点对点通信比通过中介进行通信更高效。

6.2 Online Boutique

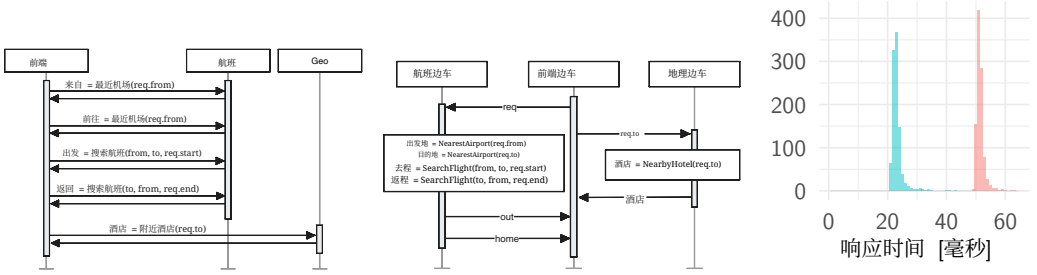
我们评估了 Online Boutique（第 2.1 节）在一个跨越 3 个可用区的 4 节点 AWS EKS 集群上的实现。此多可用区部署体现了生产级架构：服务分布在不同可用区，以通过物理隔离的基础设施增强韧性，防范相关故障，或应对单个可用区容量已满的情况 [57]。

所有实验均在 eu-north-1 区域运行。每个节点都运行一个 c5a.xlarge 实例，配备 4 个虚拟 CPU 和 8 GB 内存。每次测量前，系统以恒定的 120 请求/秒负载运行 2 分钟，以预热 Java 虚拟机。随后，负载调整为 60 请求/秒，并在 4 分钟内采集 1000 个请求。我们对原始编排器和 Accompanist 版本重复了相同的过程。我们发现，原始应用的尾部延迟波动较大，因此只考虑低于第 99 百分位的延迟。



(a) 购物车商品折线图 (b) 单区域 (c) 三个区域 （d）独立编排器

图 14. 不同部署方式下*Online Boutique* 应用中结账请求的响应时间。



（a）编排 （b）使用 Accompanist 的编舞 (c) 响应时间

图 15. 对酒店预订应用的扩展，通过远程服务预订航班。

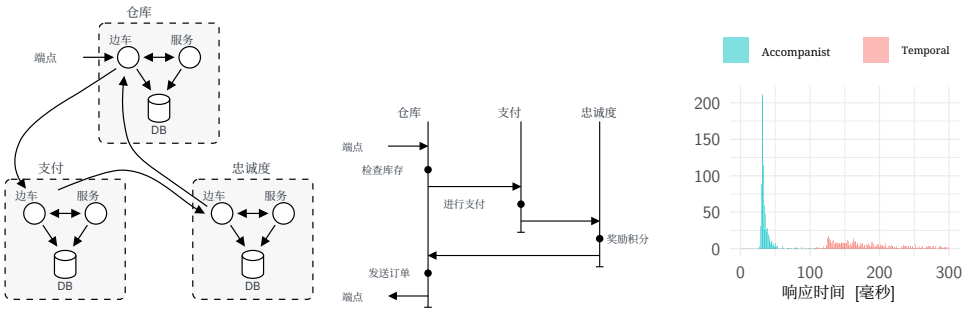
图 14a 展示了用户结账工作流的端到端响应时间，其中各服务分布在不同的可用区中。正如第 2 节所述，购物车中的商品项越多，响应时间就越长，因为每件商品都需要编排器发起两次远程过程调用。不过，Accompanist 版本的扩展性更好，因为远程过程调用由边车发起，而边车与其访问的服务 部署在同一位置 。

其余图表展示了三种部署策略下的结账延迟分布，每次均假设用户购物车中有 9 件商品。图14b 将所有服务调度到同一可用区中的两个节点上。图14c 将服务随机分布在三个可用区中的四个节点上。图14d 将所有 RPC 服务器放在一个可用区中的两个节点上，并将编排器放在另一个可用区。结果表明，当所有参与服务都能部署在编排器附近时，编排效率很高；否则，消除开销高昂的网络调用可将延迟降低多达 32%。

6.3 酒店预订

另一种常见用例是多方协作，例如分布式机票与酒店预订系统。传统上，编排器会通过广域网调用第三方提供的应用程序接口——但如第 2 节所述，第三方应用程序接口可能并未针对编排器的需求进行专门设计。因此，编排器可能需要进行多次广域网调用才能完成工作流。

我们通过改编第 2 节中的酒店预订应用来测试这一场景。该基准测试最初由一组用于搜索和预订酒店的微服务组成；我们通过新增一个用于查询机场和航班的航班微服务扩展了该基准测试。



(a) 仓库部署架构-tecture。

(b) 仓库编排。

(c) 仓库工作流请求端到端延迟。

图 16. 从仓库订购产品的 Saga 工作流。

航班服务模拟了一个第三方组织，其部署在与其他微服务不同的数据中心。我们使用编排器（图 15a）以及使用Accompanist （图15b）实现了该工作流。

航班服务单独部署在 AWS 区域 `eu-west-3` （巴黎）的 EKS 集群上，该集群由单个 `m5.large` 节点组成，配有 2 个虚拟 CPU 和 8 GB 内存。所有其他服务则一同部署在 `eu-central-1` （法兰克福）AWS 区域的 EKS 集群上，该集群由 3 个 `m5.large` 节点组成。其他方面，该基准测试的执行和测量方式与第 6.2 节所述的 Online Boutique 基准测试相同。

图 15c 展示了运行 Accompanist 和编排版本的行程计划请求时的端到端响应时间。这里，Accompanist 的平均响应时间比编排版本提升了 55%。

6.4 仓库 Saga

我们通过与成熟的编排式工作流框架 Temporal [47] 进行比较，评估 Accompanist Saga 的性能。Temporal 使用中央编排器跟踪所有正在运行的工作流，并向工作节点发送命令。这种设计通过部署更多工作节点，可以轻松实现水平扩展。这与 Accompanist 边车形成对比，后者不容易复制。Temporal 中的工作节点也采用拉取式模型，而 Accompanist 则采用推送式模型。

由于设计上的这些差异，我们预计 Accompa-nist 的端到端延迟会更低，因为服务之间采用点对点通信，而非通过中间人。轮询设计还可能导致不可预测的调度延迟，因此第6.1 节中的预测模型仅是 下界，代表 Temporal 在最坏情况下的响应能力。

作为基准，我们采用了 Temporal 自有文档中的 Saga 示例 [46] 并对其进行扩展，使其能够执行实际操作（在本地数据库中下订单），并将服务分布到多个 Kubernetes Pod 中。为便于阅读，我们还参照 Newman 的仓库 Saga 示例重新命名了这些服务 [34, 图 6-5]。该工作流由仓库、支付和忠诚度微服务组成。我们还使用 Accompanist Saga 实现了相同的工作流。图 16b展示了该编舞的时序图。

我们将两种实现都部署到了由 3 个 m5.large 节点组成、位于 eu-central-1 数据中心的 AWS EKS 集群上。通过使用 Kubernetes 亲和性规则，我们确保相关服务

被部署到同一个节点上——这对 Temporal 来说是理想场景。编舞版本的部署架构如图16a所示。

评估使用 Temporal 的生产环境默认部署配置[2]，该配置使用 Cassandra 作为数据库。这与 Accompanist 使用的 PostgreSQL 数据库具有不同的特性。我们也测试了使用 PostgreSQL 的 Temporal，但其性能不如 Cassandra。

图 16c 展示了两种实现的响应时间分布。 Accompanist的响应时间中位数为 32 毫秒，优于 Temporal 的 190 毫秒中位数，提升了 5 倍。9×。 Accompanist的尾延迟也更低，第99百分位响应时间为 87 毫秒，而 Temporal 为 587 毫秒（性能提升了 6 倍。7×）。

7 相关工作

现有的分布式编程方法在不同抽象层次上处理正确性和容错问题。我们将其分为：（i）语言级方法，通过通信和故障处理构造扩展编程模型；（ii）基于运行时的系统，通过基础设施提供容错能力；以及（iii）编排和工作流框架，用于管理长时间运行的分布式进程。这种分类突显了 Accompanist 的独特定位：它是一种面向容错编舞式程序的运行时级解决方案，能够实现去中心化 Saga 和基于重放的容错，而无需扩展源语言。

语言层方法。 大量研究通过扩展分布式系统的编程语言来解决正确性和故障处理问题。

Wiersdorf 和 Greenman [56] 研究了编舞式编程中的失效停止故障[3]，并为其 Elixir DSL Chorex 引入了一种用于处理故障的分布式 try-catch 机制。执行 try-catch 代码块时，Chorex 会创建一个检查点；当 try 代码块中的某个参与方崩溃时，该参与方会恢复到此检查点，所有参与方则转至 catch 代码块。该机制依赖于对节点崩溃的可靠检测，并依靠全局同步确保所有参与方就代码块成功还是失败达成一致，而 Accompanist 的解决方案则避免了全局同步。尽管 Chorex 没有为其恢复机制提供形式语义和形式化的正确性结果，但其 try-catch 机制与多方会话类型 [28, 51]这一相邻领域中探索的模型有一些相似之处。

Graversen 等人 [20] 研究了编舞式编程中的遗漏故障[4]，并引入语言原语，以便编程实现自定义、细粒度的故障处理和恢复策略。他们的方法表明，在语言层面可以处理某些类别的非恶意通信故障，同时保留编舞程序的关键安全保证。这些方法提供了强有力的正确性保证和富有表现力的抽象，但需要通过额外构造丰富编程模型，并要求程序员围绕故障处理逻辑（如重试、确认或超时）显式组织代码。相比之下，Accompanist 将容错移至运行时层，使现有编舞程序无需修改源语言或编译器即可容忍故障，同时保留编舞式编程特有的清晰性和可读性。

除了编舞语言之外，还有一些交互式和多层编程语言采用了这样的范式：编写单个程序，再将其编译为分布式代码 [10, 45, 52, 55]。ScalaLoci [54] 等框架通过响应式抽象扩展了这一理念，使流能够

[2]https://github.com/temporalio/helm-charts [3]在故障停止故障模式下，故障表现为节点崩溃，其余系统可以可靠地检测到此类故障。[4]当组件无法发送或接收消息，或传输层无法传送消息时，就会发生遗漏故障。

在系统各节点间共享，并沿数据流依赖关系传播故障，同时支持在每个参与方手动编写恢复策略。在 RPC 链中 [44]，多个顺序执行的远程过程调用会串联起来，然后返回给发起方，以降低延迟；同时，沿调用链发送定期心跳来检测导致调用链中断的故障。这些方法在多语言微服务开发中的应用仍然有限。我们的框架尤其适用于这一领域：通信媒介可通过 Choral 的通道 API 进行定制；类型系统会静态保证通信和数据局部性；在合理假设下，执行具有确定性；并且编译策略会自动利用并行机会。

基于运行时的方法。 基于运行时的方法通过基础设施而非语言设计来提供容错能力和可扩展性。

Erlang [5], 、Orleans [8], 和 Akka [3] 等 Actor 运行时和框架提供了灵活的抽象，用于构建分布式系统，其中包括监督层次结构和故障恢复机制。Ray [32] 等系统进一步支持分布式 future，将控制流与数据移动解耦。然而，这些抽象依赖加权引用计数 [53] 等运行时机制，在实践中要正确使用并非易事 [48–50]。尽管这些系统具有很大的灵活性，却缺少全局规范机制，也无法保证通信安全性或（全局）恢复等属性。因此，开发者必须手动确保协议正确性和恢复行为。相比之下，编舞式编程提供带有强正确性保证的全局规范，而 Accompanist 则通过运行时容错来补充这些保证。

编排、工作流与 *Saga*。 实际上，服务编舞通常使用消息代理或 RPC 库实现，也可能借助外部数据存储 [23]。尽管这些方法得到广泛采用，但它们提供的高层抽象有限，导致大型系统难以推理和维护 [33]。

为应对这些挑战，许多组织开发了编排与工作流系统 [4, 7, 9, 26, 47], ，用于集中控制分布式进程，并提供内置的容错与恢复支持。这些系统简化了对长时间运行的工作流的推理，并且通常采用 Saga 风格的执行模型，但依赖集中式编排器，可能会增加延迟、限制可扩展性，并制约（或阻止）数据局部性。

Accompanist 的架构可视为将“编排器去中心化”为一组响应式边车，在保留容错能力的同时改善端到端延迟。与传统编排系统不同，这种方法在执行期间避免了集中协调。尽管这种架构本身并不专属于编舞式编程，但编舞与之天然契合：将编排器迁移到 Choral 可以保留原程序的结构，类型系统可保证不存在通信不匹配，而编译器则确保执行具有确定性且无死锁。与集中式工作流引擎相比，Accompanist 在保持去中心化执行的同时，实现了类似的容错目标。

8 结论

我们的结果表明，Accompanist 带来的开销较小，并且在网络成本较高时比编排具有更好的扩展性。或许我们最重要的贡献，是提出将编舞式编程作为一种比远程过程调用更好的接口，让应用程序开发者能够向计算底层表达偏好。如果应用程序开发者愿意将工作流的热路径表示为编舞程序，系统研究人员便可利用这些额外信息来提升性能。一个很有前景的方向是将编舞程序集成到网络协议栈的较低层，利用网内计算等特性，并消除

支持通用远程过程调用所需的开销 [17, 40, 58]。由于编舞程序可以指定数据如何从一个位置路由到另一个位置，分析工具还可以找出如何重新路由编舞程序，以减少拥塞并优先处理流 [59]。这些问题为系统研究人员与编译器和编程语言领域的专家开展更深入的合作提供了契机。

9 数据可用性声明

Accompanist 框架的完整源代码以及如何开展评估以确保结果可复现的指令，将随工件提交一并提交。

参考文献

[1] 2025. Klor：Clojure 中的编舞。https://github.com/lovrosdu/klor。[2] Cosku Acay、Joshua Gancher、Rolph Recto 和 Andrew C. Myers。2024。分布式密码应用的安全合成。载于第 37 届 IEEE 计算机安全基础研讨会 CSF 2024，恩斯赫德，荷兰，2024年7月8日至12日。IEEE, 433 448 10 1109 61375 2024 00021 – 。https://doi.org/ 。/CSF 。 。Akka n. d. 。Akka。https://akka.io/。[3] [ ]。[4] Apache 软件基金会。 [n. d.]。Apache Airflow。https://airflow.apache.org/。[5] Joe Armstrong。2003。存在软件错误时构建可靠的分布式系统。博士论文。皇家理工学院，斯德哥尔摩，瑞典。[6] Mako Bates、Shun Kashiwa、Syed Jafri、Gan Shen、Lindsey Kuper 和 Joseph P. Near。2025。高效、可移植且适用于任意参与方数量的编舞式编程。载于 PLDI '25：第 46 届 ACM SIGPLAN 编程语言设计与实现国际会议。ACM，首尔，韩国。https://doi.org/ 10 1145 3729296 2412 02107 [ ]。/ arXiv: 。cs [7] Sebastian Burckhardt、Chris Gillum、David Justo、Konstantinos Kallas、Connor McMahon 和 Christopher S. Meik-lejohn。 [n. d.]。持久函数：有状态无服务器计算的语义。5（[n. d.]），1–27。OOPSLA 专刊。https://doi.org/ 10 1145 3485510 。/ Sergey Bykov、Alan Geller、Gabriel Kliot、James R. Larus、Ravi Pandya 和 Jorgen Thelin。 [8] 2011。Orleans：人人可用的云计算。载于第 2 届 ACM 云计算研讨会论文集——SOCC '11。ACM Press，卡斯凯什，葡萄牙，1 14 10 1145 2038916 2038932 – 。https://doi.org/ 。/ 。Camunda n. d. 。Camunda：通用流程 [9] [ ] 编排器。https://camunda.com/。[10] Roberto Casadei。2023。宏编程：宏观行为建模的概念、现状与机遇。《计算调查》 55, 13 2023 275 1 275 37 10 1145 3579353 s（7月），：–: 。https://doi.org/ 。/ [11] Philippe Charles、Christian Grothoff、Vijay A. Saraswat、Christopher Donawa、Allan Kielstra、Kemal Ebcioglu、Christoph von Praun 和 Vivek Sarkar。2005。X10：一种面向对象的非均匀集群计算方法。载于第 20 届 ACM SIGPLAN 面向对象编程、系统、语言与应用会议论文集，OOPSLA 2005，2005年10月16日至20日，美国加利福尼亚州圣迭戈，Ralph E. Johnson 和 Richard P. Gabriel （编）。ACM, 519 538 10 1145 1094811 1094852 – 。https://doi.org/ 。/ 。Vito De Giosa。n. d.。使用 AWS Step Functions 监控服务编舞。[12] [ ] https://aws.amazon.com/blogs/architecture/use-aws-step-functions-to-monitor-services-choreography/。[13] Nicola Dragoni、Saverio Giallorenzo、Alberto Lluch-Lafuente、Manuel Mazzara、Fabrizio Montesi、Ruslan Mustafin 和 Larisa Safina。2017。微服务：昨天、今天与明天。载于《当前与未来的软件工程》，Manuel Mazzara 和 Bertrand Meyer（编）。Springer，195–216。https://doi.org/10.1007/978-3-319-67425-4_12[14] Yu Gan、Yanqi Zhang、Dailun Cheng、Ankitha Shetty、Priyal Rathi、Nayan Katarki、Ariana Bruno、Justin Hu、Brian Ritchken、Brendon Jackson、Kelvin Hu、Meghna Pancholi、Yuan He、Brett Clancy、Chris Colen、Fukang Wen、Catherine Leung、Siyuan Wang、Leon Zaruvinsky、Mateo Espinosa、Rick Lin、Zhongling Liu、Jake Padilla 和 Christina Delimitrou。2019。用于微服务及其对云与边缘系统软硬件影响分析的开源基准测试套件。载于第 24 届编程语言与操作系统架构支持国际会议论文集。ACM，美国罗得岛州普罗维登斯， 3 18 10 1145 3297858 3304013 – 。 [15] https://doi.org/ 。/ 。Hector Garcia-Molina 和 Kenneth Salem。 1987 。Saga。载于 ACM SIGMOD 数据管理国际会议论文集（美国加利福尼亚州旧金山）（SIGMOD '87）。计算机协会，美国纽约州纽约， 1987 249 259 10 1145 38713 38742 – 。https://doi.org/ 。/ 。Sanjay Ghemawat、Robert Grandl、 [16] Srdjan Petrovic、Michael Whittaker、Parveen Patel、Ivan Posva 和 Amin Vahdat。2023。迈向云应用的现代化开发。载于第 19 届操作系统热点主题研讨会论文集，HOTOS 2023，美国罗得岛州普罗维登斯，2023年6月22日至24日，Malte Schwarzkopf、Andrew Baumann 和 Natacha Crooks（编）。ACM，110–117。https://doi.org/10.1145/3593856.3595909

[17] Saverio Giallorenzo、Jacopo Mauro、Andrea Melis、Fabrizio Montesi、Marco Peressotti 和 Marco Prandini。2025。编排定义网络：拒绝服务缓解案例研究。载于《面向服务的计算》，Walid Gaaloul、Michael Sheng、Qi Yu 和 Sami Yangui（编）。第 15405 卷。Springer Nature Singapore，新加坡，243–259。https://doi.org/10 1007 978 981 96 0808 9_18 。/ - - - - Saverio Giallorenzo、Fabrizio Montesi 和 Marco Peressotti。2024。[18] Choral：面向对象的编舞程序。ACM《编程语言与系统汇刊》46, 1 2024 1 1 1 59 10 1145 3632398 （），：–：。https://doi.org/ 。/ [19] Google Cloud Platform。2024。。GoogleCloudPlatform/Microservices-Demo。https://github.com/GoogleCloudPlatform/microservices-demo。[20] Eva Graversen、Fabrizio Montesi 和 Marco Peressotti。2025。编舞式编程中的遗漏故障。载于第 26 届意大利理论计算机科学会议论文集，佩斯卡拉，意大利，2025年9月10日至12日（CEUR 研讨会论文集，第 4039 卷），Luca Moscardelli 和 Francesca Scozzari（编）。CEUR-WS.org，169–184。[21] gRPC Authors。[n. d.]。gRPC。https://grpc.io/。[22] Thomas Heinrichs。[n. d.]。编排与编舞。https://camunda.com/blog/2023/02/orchestration-vs-choreography/。[23] Gregor Hohpe 和 Bobby Woolf。2004。《企业集成模式：消息解决方案的设计、构建与部署》。Addison-Wesley，波士顿。[24] Fan Lai、Jie You、Xiangfeng Zhu、Harsha V. Madhyastha 和 Mosharaf Chowdhury。2020。Sol：慢速网络上的快速分布式计算。载于第 17 届 USENIX 网络系统设计与实现研讨会，NSDI 2020，美国加利福尼亚州圣克拉拉，2020年2月25日至27日，Ranjita Bhagwan 和 George Porter（编）。USENIX 协会，273–288。[25] Tanakorn Leesatapornwongsa、Jeffrey F. Lukman、Shan Lu 和 Haryadi S. Gunawi。2016。TaxDC：数据中心分布式系统中非确定性并发错误的分类体系。载于 ASPLOS。ACM，517–530。[26] LogicApps [n. d.]。Azure Logic Apps文档。https://learn.microsoft.com/en-us/azure/logic-apps/。[27] Lovro Lugović 和 Fabrizio Montesi。2024。现实世界中的编舞式编程：全双工异步与互操作性。Art Sci. Eng. Program. 8, 2 2024 10 22152 2024 8 8 （）。[28] 2025 https://doi.org/ 。/PROGRAMMING-JOURNAL.ORG/ / / Claudio Antares Mezzina、Francesco Tiezzi 和 Nobuko Yoshida。。会话编程中的基于检查点的回滚恢复。《计算机科学逻辑方法》第 21 卷，第 1 期（2025年1月）。https://doi.org/10.46298/lmcs-21(1: 2 2025) Microsoft Azure。2025。边车模式。[29] https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar。[30] Fabrizio Montesi。2014。编舞式编程。博士论文。哥本哈根信息技术大学，丹麦。[31] Fabrizio Montesi。2023。《编舞简介》（第 1 版）。剑桥大学出版社，剑桥。[32] Philipp Moritz、Robert Nishihara、Stephanie Wang、Alexey Tumanov、Richard Liaw、Eric Liang、Melih Elibol、Zongheng Yang、William Paul、Michael I. Jordan 和 Ion Stoica。2018。Ray：面向新兴人工智能应用的分布式框架。arXiv: 1712 05889 [ ] 2018 1712 05889 [ ]。cs, stat（9月）。arXiv: 。cs, stat Netflix Technology Blog。2016。。Netflix Conductor：微服务编排器。https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40。[34] Sam Newman。2021。《构建微服务：细粒度系统的设计》（第二版）。O’Reilly，波士顿、法纳姆、塞巴斯托波尔、东京、北京。[35] Oracle。2025。多云已成新常态。https://www.oracle.com/cloud/multicloud/mainstream。[36] Hojin Park、Ziyue Qiu、Gregory R. Ganger 和 George Amvrosiadis。2024。利用自动配置的 MACARON缓存降低跨云／跨区域成本。载于 ACM SIGOPS 第 30 届操作系统原理研讨会论文集，SOSP 2024，美国得克萨斯州奥斯汀，2024年11月4日至6日，Emmett Witchel、Christopher J. Rossbach、Andrea C. Arpaci-Dusseau 和 Kimberly Keeton（编）。ACM，347 368 10 1145 3694715 3695972 – 。https://doi.org/ 。/ 。[37] Dan Plyukhin、Marco Peressotti 和 Fabrizio Montesi。2024。。Ozone：完全乱序编舞。载于第 38 届欧洲面向对象编程会议，ECOOP 2024，2024年9月16日至20日，奥地利维也纳（LIPIcs，第 313 卷），Jonathan Aldrich 和 Guido Salvaneschi（编）。Schloss Dagstuhl - Leibniz信息学中心，31:1–31:28。https://doi.org/10 4230 2024 31 。/LIPICS.ECOOP. 。Morgan Quigley、Ken Conley、Brian Gerkey、[38] Josh Faust、Tully Foote、Jeremy Leibs、Rob Wheeler、Andrew Y Ng 等。2009。ROS：开源机器人操作系统。载于 ICRA 开源软件研讨会，第 3 卷。神户，5。[39] Chris Richardson。2019。《微服务模式：附 Java 示例》。Manning Publications。[40] Divyanshu Saxena、William Zhang、Madhav Tummala、Saksham Goel 和 Aditya Akella。2023。特邀论文：迈向高效的微服务通信。载于第 5 届分布式系统算法实现与评估高级工具、编程语言及平台研讨会论文集。ACM，美国佛罗里达州奥兰多，1–5。https://doi.org/10 1145 3584684 3597267 。/ 。Korakit Seemakhupt、[41] Brent E. Stephens、Samira Manabi Khan、Sihang Liu、Hassan M. G. Wassel、Soheil Hassas Yeganeh、Alex C. Snoeren、Arvind Krishnamurthy、David E. Culler 和 Henry M. Levy。2023。远程过程调用的云规模特征分析。载于第 29 届操作系统原理研讨会论文集，SOSP 2023，科布伦茨，

德国，2023 年 10 月 23–26 日，Jason Flinn、Margo I. Seltzer、Peter Druschel、Antoine Kaufmann 和 Jonathan Mace（编）。ACM，498 514 10 1145 3600006 3613156 – 。https://doi.org/ 。/ 。Gan Shen、Shun Kashiwa 和 Lindsey Kuper[42]。。HasChor：面向所有人的函数式编排式编程（函数式明珠）。《ACM 编程语言会议录》2023 7， 2023 541 565 10 1145 3607849 ICFP（），– 。https://doi.org/[43]2025。/ Shopify。。Toxiproxy。https://github.com/Shopify/toxiproxy。[44] Yee Jiun Song、Marcos Kawazoe Aguilera、Ramakrishna Kotla 和 Dahlia Malkhi。2009 年。RPC 链：地理分布式系统中的高效客户端-服务器通信。载于《第六届 USENIX 网络系统设计与实现研讨会论文集》（NSDI 2009），2009 年 4 月 22–24 日，美国马萨诸塞州波士顿，Jennifer Rexford 和 Emin Gün Sirer（编）。USENIX 协会，277–290。http://www.usenix.org/events/nsdi09/tech/full_papers/song/song.pdf[45] Joskel Ngoufo Tagueu、Adrián Puerto Aubel 和 Maurice Tchoupe Tchendji。2025 年。用于增量计算的服务组合引擎。载于《计算机科学研究》，Djamil Aissani、Kamel Barkaoui 和 Mathieu Roche（编）。施普林格·自然瑞士，查姆，156–171。[46] Temporal Technologies。2025 年。订单 Saga 示例。https://github.com/temporal-sa/temporal-order-saga。[47] Temporal Technologies。2025 年。Temporal.io。https://temporal.io/。[48] Ray 团队。2023 年。反模式：在循环中调用 Ray.Get 会损害并行性。https://docs.ray.io/en/latest/ray- core/patterns/ray-get-loop.html。[49] Ray 团队。2023 年。反模式：不必要地调用 Ray.Get 会损害性能。https://docs.ray.io/en/latest/ray- core/patterns/unnecessary-ray-get.html。[50] Ray 团队。2023 年。反模式：使用 Ray.Get 按提交顺序处理结果会增加运行时。https: // docs.ray.io/en/latest/ray-core/patterns/ray-get-submission-order.html。[51] Malte Viering、Raymond Hu、Patrick Eugster 和 Lukasz Ziarek。2021 年。一种用于容错事件驱动分布式编程的多方会话类型纪律。《ACM 编程语言会议录》5，OOPSLA（2021），1–30。https: //doi.org/ 10 1145 3485501 。/ Mattias De Wael、Stefan Marr、Bruno De Fraine、Tom Van Cutsem 和 Wolfgang De Meuter[52]。2015 年。分区全局地址空间语言。《计算调查》47, 4 2015 62 1 62 27 10 1145 2716320 （），：–：。https://doi.org/[53] 。/ Stephanie Wang、Eric Liang、Edward Oakes、Benjamin Hindman、Frank Sifei Luan、Audrey Cheng 和 Ion Stoica。2021 年。Ownership：一种面向细粒度任务的分布式 Future 系统。载于《第 18 届 USENIX 网络系统设计与实现研讨会》（NSDI 2021），2021 年 4 月 12–14 日，James Mickens 和 Renata Teixeira（编）。USENIX 协会，671–686。[54] Pascal Weisenburger、Mirko Köhler 和 Guido Salvaneschi。2018 年。使用 ScalaLoci 开发分布式系统。《ACM 编程语言会议录》2, 2018 1 30 10 1145 3276499 OOPSLA（[55]2020 10 月），– 。https://doi.org/ 。/ Pascal Weisenburger、Johannes Wirth 和 Guido Salvaneschi。。多层编程综述。《计算调查》53, 4 2020 1 35 10 1145 3397495 （9 月），– 。https://doi.org/ 。/ Ashton Wiersdorf[56] 和 Ben Greenman。。Chorex：可重启的语言集成式编排。《编程的艺术、科学与工程》10，3（2025 年 10 月），20:1–20:30。https://doi.org/10.22152/programming-journal. org/ 2025 10 20 / / Marcus Young。2015 年。为 AWS 实施云设计模式：在 AWS 云中创建高效的可扩展性、冗余和高[57]可用性设计模式（第 1 版）。Packt 出版社，英国伯明翰。[58] Xiangfeng Zhu、Weixin Deng、Banruo Liu、Jingrong Chen、Yongji Wu、Thomas Anderson、Arvind Krishnamurthy、Ratul Mahajan 和 Danyang Zhuo。2023 年。应用定义网络。载于《第 22 届 ACM 网络热点问题研讨会论文集》。ACM，美国马萨诸塞州剑桥，87–94。https://doi.org/10.1145/3626111.3628178[59] Yibo Zhu、Haggai Eran、Daniel Firestone、Chuanxiong Guo、Marina Lipshteyn、Yehonatan Liron、Jitendra Padhye、Shachar Raindel、Mohamad Haj Yahia 和 Ming Zhang。2015 年。大规模 RDMA 部署的拥塞控制。载于《2015 年 ACM 数据通信特别兴趣小组会议论文集》。ACM，英国伦敦，523–536。https://doi.org/10.1145/2785956.2787484