Practical Byzantine Fault Tolerance and Proactive Recovery

我们对互联网上可访问的在线服务日益增长的依赖性需要高可用性的无需中断即可提供正确服务的系统。软件漏洞，运营商错误和恶意攻击是服务中断的主要原因，而且它们可能导致任意行为，也就是拜占庭故障。本文介绍了一种新的复制算法 BFT，可用于构建容忍拜占庭故障的高可用性系统。BFT 可以在实践中用于实现真实的服务：它表现良好，在异步环境（如互联网）中是安全的，它包含了防御拜占庭故障客户端的机制，并且它可以主动恢复副本。这允许复制系统在有少于 1 / 3 的副本在攻击窗口（window of vulnerability）产生故障的系统的生命周期内容忍任何数量的故障。BFT 已经被实现为具有简单接口的通用程序库。我们使用这个库实现了第一个支持 NFS 协议的拜占庭容错分布式文件系统，BFS。BFT 库和 BFS 表现良好，因为该库包含了几个重要的优化，其中最重要的是使用对称加密来验证消息。性能结果表明，BFS 的性能与未复制的 NFS 协议的产品实现相比，范围在快 2％到慢 24％之间。这支持了我们对于 BFT 库可用于实现实际的拜占庭容错系统的观点。

General Terms: Security 安全性, Reliability 可靠性, Algorithms 算法, Performance 性能, Measurement 度量

Additional Key Words and Phrases: Byzantine fault tolerance 拜占庭容错, state machine replication 状态机复制, proactive recovery 主动恢复, asynchronous systems 异步系统, state transfer 状态转变

1. Introduction

我们越来越依赖计算机系统提供的服务，然而我们的计算机故障的脆弱性也在不断增加。我们希望这些系统是高度可用的：它们应该正常工作，并且它们应该在不中断的情况下提供服务。

有大量关于实现高可用性系统的复制技术的研究。问题是，大部分关于复制技术的研究着重于容忍良性故障（比如，Alsberg and Day [1976], Gifford [1979], Oki and Liskov [1988], Lamport [1989], and Liskov et al. [1991]）：这些技术假设组件通过停止或省略某些步骤而失败。如果有一个故障组件违反了该假设，它们就可能不会提供服务。不幸的是，因为恶意攻击、运营商错误以及软件错误是故障产生的通常原因，而且它们会导致故障节点表现出任意的行为，也就是拜占庭故障，所以这个假设是无效的。工业和政府对计算机系统的日益增长的依赖提供了恶意攻击的动机，而日益增加的网络连接将这些系统暴露在更多攻击的面前。运营商的错误也被认为是导致失败的主要原因[Murphy and Levidow 2000]。此外，由于软件的规模和复杂性的增长，软件错误的数量也在增加。

拜占庭容错技术[Pease et al. 1980; Lamport et al. 1982]为这个问题提供了一个潜在的解决方案，因为他们对故障进程的行为没有做出任何假设。在可以容忍拜占庭故障的一致性和复制技术方面，已经有大量的工作成果。然而，大部分简单的工作（比如，Canetti and Rabin [1992], Reiter [1996], Malkhi and Reiter [1996b], Garay and Moses [1998], and Khilstrom et al. [1998]）即没有考虑到技术在实际应用中效率会很低，也没有依据这样的假设，它们被攻击之后会轻易地变得无效。举个例子，在互联网中依赖同步性（synchrony）[Lamport 1984]是非常危险的，也就是依赖于消息延迟和进程速度的边界。攻击者可以通过延迟非故障节点或它们之间的通信来破坏服务的正确性，直到超出边界。这样的拒绝服务攻击（denial-of-service attack）通常比控制非故障节点容易得多。

本文介绍了 BFT，一种新的状态机复制算法[Lamport 1978; Schneider 1990]，在共有 n 个副本中最多[(n−1)/3]是故障的情况可以提供活性和安全性。这意味着客户端最终会收到他们的请求的回复，而且根据可线性化性证明这些回复是正确的[Herlihy and Wing 1987; Castro and Liskov 1999a]。

BFT 是第一个在异步系统（比如互联网）中，拜占庭容错的状态机复制算法：它不需要依赖任何同步假设来提供安全性。尤其是，即使在拒绝服务供给下也不会返回错误的回复。另外，它保证了提供活性的消息延迟最终是有界的。当拒绝服务攻击发生的时候，服务端可能不会返回回复，但是客户端在攻击结束的时候保证会收到回复。

因为 BFT 是一个状态机复制算法，它有能力以复杂的操作来复制服务。这是一个重要的对拜占庭故障客户端的防御手段：操作可以设计为在服务状态上保留不变量，提供窄接口，并执行访问权限控制。无论故障客户端数量多少，BFT 都提供安全性，安全性确保故障客户端无法中断这些不变量或绕过访问权限控制。只提供读、写和同步原语的算法（例如，Malkhi and Reiter [1998b]）更容易受到拜占庭客户端的影响；它们依赖于客户端来正确地命令和同步读写，以便强制执行不变量。

我们还介绍了一种积极的 BFT 恢复机制，即使没有理由怀疑它们有故障，它也会周期性地恢复复制。这允许复制系统在有少于 1 / 3 的副本在攻击窗口（window of vulnerability）产生故障的系统的生命周期内容忍任何数量的故障。如果在一个系统的生命周期中少于 1 / 3 的副本失败，那么之前可以保证的最好的就是正确的表现。在对性能只有一点点影响的正常情况下，攻击窗口可以非常短（比如几分钟）。我们的机制提供了拒绝服务攻击的检测，旨在增加窗口时长，同时还可以检测一个副本的状态是否被攻击者损坏了。

BFT 包含了一些重要的优化，允许算法有更好的性能，从而可以在实践中使用它。最重要的优化是使用对称加密来验证消息。公共密钥加密技术，在之前的系统里被称为主要延迟[Reiter 1994]和吞吐量[Malkhi and Reiter 1996a]，仅用于交换对称密钥。其他优化减少了通信开销：该算法只使用一个消息回路执行只读操作，两个回路执行读写操作，并且在负载下使用批处理，以在许多请求上为读写操作分摊协议开销。该算法还使得随着操作参数和结果大小增加而减少协议开销来优化。另外，本文介绍了有效的技术来回收协议信息垃圾，并转移状态以使副本更新；这些都是建立容忍拜占庭故障的实际服务所必需的。

BFT 已经被实现为具有简单接口的通用程序库。BFT 库可用于提供不同服务的拜占庭容错版本。本文介绍了 BFT 库，并解释了它如何用于实现实际的服务：第一个支持 NFS 协议的拜占庭容错分布式文件系统，BFS。

本文介绍了 BFT 库和 BFS 的性能分析。实验结果表明，BFS 的性能与未复制的 NFS 协议的产品实现相比，范围在快 2％到慢 24％之间。结果从有四个和七个可以分别容忍一个和两个拜占庭故障的副本的配置中获得。他们支持了我们对于 BFT 库可用于实现实际的拜占庭容错系统的观点。

本文其余部分内容安排如下。第二节说明了我们系统的模型和假设，第三节描述了通过该算法和状态正确性条件解决的问题。第四节非正式地描述了没有恢复的算法，而正式的描述放在了附录中。第五节介绍了主动恢复机制。第六节描述了在出现拜占庭故障时实现复制的实用解决方案的重要优化办法和实现技术。BFT 库和 BFS 的实现在第七节中给出。第八节对 BFT 库和 BFS 进行了详细的性能分析。第九节讨论相关工作。最后，我们的结论和未来工作的一些方向将在第十节中出现。

2. SYSTEM MODEL

复制服务由 n 个执行客户请求的操作的副本实现。副本和客户端在分布式系统中的不同节点上运行，并通过网络连接。

BFT 实现了一种允许对执行给定是确定性的任意计算的服务进行复制的状态机复制的形式[Lamport 1978; Schneider 1990]，也就是说，当它们处理相同的操作序列时，复制必须产生相同结果的序列。

副本使用一个加密的哈希函数 D 来计算消息摘要（message digests），他们使用消息认证码（MACs）来对所有包括客户端请求[Schneier 1996]的消息进行身份验证。每对副本 i 和 j 都有一对会话密钥：$k_{i,j}$ 用于计算从 i 到 j 发送的消息的 MACs，$k_{j,i}$ 用于从 j 发送到 i 的消息。每个副本也和每个客户端共享一个秘密秘钥；这个秘钥用来验证通信的所有两个方向。这些会话密钥可以使用第 5.2.2 节或任何其他密钥交换协议中描述的机制，动态地建立和刷新。

点对点发送到单个收件人的消息包含单个 MAC；我们将这样的消息表示为$\langle m \rangle_{\mu_{ij}}$，其中 i 是发送方，j 是接收方，并且使用 $k_{i,j}$ 来计算 MAC。多播到所有副本的消息包含认证器（authenticators）；我们把这样一个消息表示为$\langle m \rangle_{\alpha_i}$，其中 i 是发送方。一个认证器是 MACs 的一个向量，每个副本 j（$j \neq i$）拥有一个，其中使用 $k_{i,j}$ 计算条目 j 中的 MAC。消息的接收者通过检查认证器中的相应 MAC 来验证其真实性。

BFT 从节点和网络中占用的特别少。我们使用一个拜占庭故障模型；也就是说，故障节点可能表现的反复无常。（如果副本和客户端遵循第四节中的算法，它们则是正确的。）连接节点的网络可能无法传递消息，延迟消息，复制消息，或者乱序传递消息。因此，我们允许一个非常强大的对手的存在，它可以控制故障节点和网络，以便对复制的服务造成最大的损害。例如，它可以协调故障节点、延迟消息或注入新消息。

我们只依赖于以下假设：前两个假设是关于节点行为的假设，既要求安全性，也要求活性，最后一个假设是关于网络行为的假设，只要求活性。主动恢复机制依赖于第 5.1 节中描述的额外的（现实的）假设。

Bound on Faults

我们对故障副本的数量假定一个范围 f = [(n-1) / 3]。在第 5 节中，我们描述了一种主动恢复机制，使得该算法在任何短时攻击窗口中最多有 f 个副本失败的情况下，能够容忍系统的生命周期内的任何数量的故障。但主动恢复机制需要额外的假设条件。

当在副本的故障概率之间存在强烈的正相关性时，使用 BFT 库或者任何其他复制技术是没有什么好处的；在这种情况下，违反故障数量范围的概率并不明显大于单一故障的概率。举个例子，我们的方法不能掩盖在所有副本上同时发生的软件错误。但是，BFT 库可以掩盖非确定性软件错误，这似乎是最持久的[Gray 2000]，因为它们是最难发现的。

通过采取措施以增加多样性，可以进一步增加复制的收益。一种可能性是在执行环境中增加多样性：副本可以由不同的人管理；它们可以在不同的地理位置上；并且它们可以具有不同的配置（例如，运行不同的服务组合或者运行具有不同参数的调度器）。这提高了对几种类型故障的恢复能力，比如管理员攻击或错误，涉及对副本进行物理访问的攻击，利用其他服务中的弱点的攻击以及由于竞争条件引起的软件错误。另一种可能是拥有软件多样性：副本可以运行不同的服务实现，以提高对软件漏洞和利用软件漏洞进行的攻击的恢复力。本

文中描述的 BFT 库的版本不允许软件多样性，但我们最近对库开发了一个允许软件多样性的扩展[Rodrigues et al. 2001]。

Strong Cryptography

我们还假设对手从计算方面上讲是有界的，因此（有非常高的概率）它无法推翻上面提到的加密技术。我们假设攻击者不能伪造 MACs：如果 i 和 j 是非故障节点，并且从未生成 <m>$_{\mu ij}$，则对手无法为任何 m 生成<m>$_{\mu ij}$。我们还假设加密哈希函数是抗冲突的：对手不能发现两个不同的消息 m 和 m’，使得 D(m) = D(m’)。这些假设是概率性的，但是对于我们使用的加密原语，它们被认为具有很高的概率[Black et al. 1999; Rivest 1992]。因此，我们假设在其余的文本中概率为 1。

该算法不依赖于附加到消息的任何形式的加密签名来证明它们对第三方是可信的。因此，可以很容易地对它进行修改，使其仅依赖点对点的已认证通道。这可以简单地通过多通道发送消息的拷贝（不需要 MACs）而不是多播消息（使用 MACs）来完成。通过使用消息的值来替换消息的散列而不使用加密哈希函数来修改该算法也是可能的。所得到的算法对于不具有计算上的范围的对手是安全的，倘若认证通道可以在面对这种对手时变得安全（这可能使用比如量子密码学[Bennett et al. 1992]）。但是由于大多数认证的通道实现依赖于对手的计算上的范围，所以我们提出了依赖于这个假设的算法的有效版本。

另外，如果我们只关心非恶意的故障（例如软件错误），则可以放宽关于加密原语的假设，并使用更弱，更有效的结构。

Weak Synchrony (Only for Liveness)

令 delay(t)是第一次发送消息的时刻 t 和它的目的地接收到的时刻（发送方持续重发消息直到被接收到，而且发送方和目的地都是正确的）的时间间隔。我们假设 delay(t)有一个渐近的上界。目前，我们假设 delay(t) = o(t)，但边界函数可以很容易地改变。

3. SERVICE PROPERTIES

假设在系统的整个生命周期内，不超过[(n-1)/3]个副本是有错误的，BFT 既提供了安全性和活性[Lamport 1977]。

安全性是一种线性化的形式[Herlihy and Wing 1987]：复制服务作为一个集中化的实现，一次一个原子地执行操作。线性化的原始定义对拜占庭故障客户端无效。我们在附录 B 中描述了我们修改的线性化的定义。

BFT 的弹性是最佳的：至少 3f + 1 的副本是必要的，以提供在我们的假设条件下（当最多 f 个副本出现故障时）的安全性和活性。为了理解故障副本数量的限制，可以考虑一个复制服务，该服务实现一个具有读写操作的可变变量。为了提供活性，服务可能必须要在超过 n - f 个副本接收请求之前返回一个回复，因为 f 个副本可能是有故障的并且无响应的。因此，在将新值仅写入具有 n - f 个副本的集合 W 之后，服务可以回复一个写请求。如果稍后客户端发出读请求，它可以接收一个基于具有 n - f 个副本的集合 R 的状态的回复。R 和 W 可能只有 n - 2f 个共同的副本。另外，没有响应的 f 个副本可能没有故障，因此，那些响应的 f 个副本可能是有故障的。结果，R 和 W 之间的交点可能只包含 n - 3f 个无故障副本。除非 R 和 W 至少有一个共同的非故障副本，否则无法确保读取返回正确的值；因此 n > 3f。

无论客户端有多少故障客户端在使用服务（即使它们与错误的副本共存），都会提供安全性：故障客户端执行的所有操作都会以一致的方式被无故障客户端观察到。特别地，如果服务操作被设计为在服务状态上保留一些不变量，则故障客户端不能打破这些不变量。这是对通过 BFT 实现任意抽象数据类型[Liskov and Zilles 1975]的能力启用的拜占庭故障客户端的重要防御手段。

一些算法只提供原语来读取单个变量或写入单个变量；他们更容易受到拜占庭故障客户端的影响，因为他们依赖于客户端使用这些原语来实现复杂的服务操作。即使系统提供互斥的操作来对读写进行分组（例如，Malkhi and Reiter [1998b, 2000]），它们依赖客户端来正确地排序和分组这些原语操作，以执行这些服务操作所需的不变量。举个例子，创建一个文件需要更新元数据信息。在 BFT 中，可以实现这个操作来执行元数据不变量，比如确保文件被分配给一个新的索引节点（inode）。在依靠客户端实现复杂操作的算法中，一个故障客户端将能够写入元数据信息并违背重要的不变量；比如，它可以将另一个文件的索引节点分配给新创建的文件。

由服务操作执行的不变量可能不足以防范有故障的客户端；比如说，在文件系统中，故障客户端可以将垃圾数据写入某些共享文件。因此，我们通过提供访问控制进一步限制一个故障客户端可以造成的损坏量：如果发出请求的客户端没有权利调用该操作，我们会验证客户端并拒绝访问。由于操作可以是任意复杂的，访问控制策略可以在一个抽象级别上（例如，在目录中创建文件的能力）来指定。这与访问控制策略只能指定读取或写入每个对象的能力的系统（例如，Malkhi and Reiter [1998b，2000]）形成对比。此外，该算法允许服务在保持线性化的同时动态地更改访问权限。这提供了一种从故障客户端的攻击中恢复的机制。

BFT 不依赖同步来提供安全性。因此，它必须依赖同步来提供活性；否则可以用于在异步系统中实现共识算法，但这是不可能的[Fischer et al. 1985]。倘若最多有[(n-1) / 3]个副本有故障，而且 delay(t)不会无限期的比 t 增长地更快，我们可以保证活性（即，客户端最终收到他们的请求的答复）。这是一个相当弱的、在任何真实的系统中都可能是正确的同步假设，如果网络故障最终被修复、拒绝服务攻击最终停止，而且它使我们能够规避不可能的结果。

我们的算法没有解决容错隐私问题：错误的副本可能会泄漏信息给攻击者。在一般情况下提供容错隐私尚不实际，因为服务操作可以使用其参数和服务状态来执行任意计算；副本需要这些信息才能有效地执行这些操作。但是通过让客户端加密对服务操作不透明的参数，则很容易确保隐私。

可以容忍拜占庭故障的算法是非常微妙的。因此，正式地说明和证明它们的正确性就是非常重要的。我们为简化版本的算法编写了一个正式的规范，并证明了其安全性[Castro 2001]。除了消息使用公钥加密进行身份验证以外，简化的版本与本文中描述的版本相同。最近，Lampson [2001]正式确定了本文中描述的算法的简化版本（没有公钥加密），并且讨论了它的正确性。

4. THE BFT ALGORITHM

本节介绍了无需主动恢复的算法。我们省略了与消息重传有关的一些重要的优化和细节。优化在第 6 节中解释，消息重传在 Castro [2001]中进行了说明。我们在附录中介绍了正式化的算法。

4.1. Overview

我们的算法建立在以前的状态机复制算法的工作上[Lamport 1978; Schneider 1990]。该服务被建模为在分布式系统中跨不同节点复制的状态机。每个副本维护服务状态并实现服务操作。客户端向副本发送执行操作的请求，BFT 确保所有非故障副本以相同的顺序执行相同的操作。由于副本是确定性的并且处于相同状态，因此所有非故障副本都会为每个操作发送具有相同结果的回复。客户端等待来自具有相同结果的不同副本的 f + 1 个回复。由于这些副本中至少有一个是没有错误的，所以这是操作的正确结果。

状态机复制算法中的难题是确保非故障副本以相同的顺序执行相同的请求。像 Viewstamped Replication [Oki and Liskov 1988]和 Paxos [Lamport 1989]，我们的算法使用主要备份（primary-backup）[Alsberg and Day 1976]和 quorum replication [Gifford 1979]技术的组合来排序请求。但它容忍拜占庭故障，而 Paxos 和 Viewstamped Replication 仅容忍良性故障。

在主要备份机制中，复制通过一系列称为视图（views）的配置移动。在一个视图中，一个副本是主副本（primary），其他的是备份（backups）。Primary 选择执行客户端请求的操作的顺序。它通过将下一个可用的序列号分配给请求并将此分配发送到 backups 以实现。但 primary 可能是错误的：它可以为不同的请求分配相同的序列号、停止分配序列号、或在序列号之间留下间隔（gap）。因此，备份会检查 primary 分配的序列号并使用超时值来检测何时停止。它们触发 view 的变更以在显示当前的 primary 出现故障时选择新的 primary。

该算法确保请求序列号是密集的（dense），即没有序列号被跳过，但是当有视图更改时，可能会将一些序列号分配给执行为无操作的空请求。

要正确地排序请求而不管故障，我们需要依赖 quorums [Gifford 1979]。我们可以使用任何拜占庭传播 quorum 制度架构[Malkhi and Reiter 1998a]。这些 quorums 有两个重要的属性。

—Intersection：任何两个 quorums 都至少有一个共同的正确副本。

—Availability：始终存在一个没有错误副本的可用 quorum。

这些属性使得可以使用 quorums 作为协议信息的可靠内存。副本将信息写入一个 quorum，并收集 quorum certificates，这些证书是来自一个表示它存储了信息的 quorum 中每个元素的一条消息的集合。这些证书被证明信息已被可靠地存储并将在稍后的读取中反射。从可靠的内存的读取获得由在一个 quorum 中所有元素存储的信息，并选取最新的信息。

我们也使用 weak certificates，这些证书是包含来自不同副本的至少 f + 1 个消息的集合。Weak certificates 证明至少有一个正确的副本存储了信息。协议的每一步都被一个 certificate 所证明。

我们用 R 表示复本集，并使用{0，...，| R | - 1}中的整数标识每个副本。为简单起见，我们假设| R | = 3f + 1 其中 f 是可能有故障的副本的最大数量。我们选择视图的 primary 为副本 p，使得 p = v mod | R |，其中 v 是视图编号，视图是连续编号的。目前，我们的 quorum 只是一组至少有 2f + 1 个副本的集合。

4.2. The Client

客户端 c 通过向副本多播一条<REQUEST，o，t，c>$_{\alpha c}$消息来请求执行状态机操作 o。时间戳 t 用于确保客户端执行恰好一次语义。以便后来的请求具有比更早的请求更高的时间戳，c 的请求的时间戳被完全排序。

副本接受请求并将其添加到它们的日志中，前提是它们可以对其进行身份验证。请求执行使用下一节中描述的协议进行排序。副本将请求的回复直接发送给客户端。答复的格式为<REPLY, v, t, c, i, r>$_{\mu ic}$，其中 v 是当前视图编号，t 是相应请求的时间戳，i 是副本号，r 是执行请求操作的结果。

在接受结果 r 之前，客户端等待一个来自不同副本的有 f + 1 个带有有效 MACs 的以及相同的 t 和 r 的 weak certificate，由于至多 f 个副本可能有故障，因此这确保了结果有效。我们将此认证称为 reply certificate。

如果客户端没有足够迅速的收到一个 reply certificate，则会重新发送请求。如果请求已被处理，则副本只是重新发送回复；副本记住他们发送给每个客户端的最后一个回复消息以启用此重传。如果 primary 没有为请求分配有效的序列号，则最终将被足够多的副本怀疑为有故障的，以导致视图变更。

我们假设客户端在等待一个请求完成之后再发送下一个请求，但是不难更改协议以允许客户端进行异步请求，还保留对它们的排序约束。

下面的段落讨论了客户端数量的可扩展性。首先，副本与每个客户端共享一个密钥。这可能会导致大量客户端的可扩展性问题。我们通过如下方法来避免这个问题。副本只与活跃客户端共享密钥，并限制活跃客户端的数量。当活跃客户端组更改时，可以按第 5.2.2 节所述建立新的会话密钥。即使对活跃客户端的数量有很大的范围，密钥信息也不会占用大量的空间。例如，带有 50000 个活跃客户端的信息使用少于 1 MB 的空间，假设使用 16 字节密钥和 8 字节客户端标识符。

此外，副本需要记住每个客户端执行的最后一个请求的 8 字节时间戳以确保恰好只有一次语义。但是由于时间戳很小，而且不活跃的客户端的时间戳可以存储在磁盘上，所以这不应该引起明显的可扩展性问题。然而，副本还存储发送到每个客户端的最后回复消息以启用重传。如果回复量很大，并且有大量客户端，这其实是不切实际的。这个实现可以权衡能力来为可扩展性重发丢失的回复消息。副本可以通过丢弃最早的回复来限制用于存储此信息的空间量。如果副本接收到其答复已被丢弃的请求，则通知客户该请求已被执行，但该回复不再可用。我们认为，可以使得请求重传的范围和频率足够大是不太可能发生的。此外，客户端能够在发生这种情况后查询服务并获得回复。

4.3. Normal Case Operation

我们使用三相协议来将请求原子地多播到副本。这三个阶段分别是预准备（pre-prepare）、准备（prepare）和提交（commit）。即使在提出请求顺序的 primary 是错误的情况下，pre-prepare 和 prepare 也被用来完全排序在同一个视图中发送的请求。准备和提交阶段用于确保提交的请求在整个视图中被完全排序。图 1 提供了在无故障的正常情况下算法的概述。

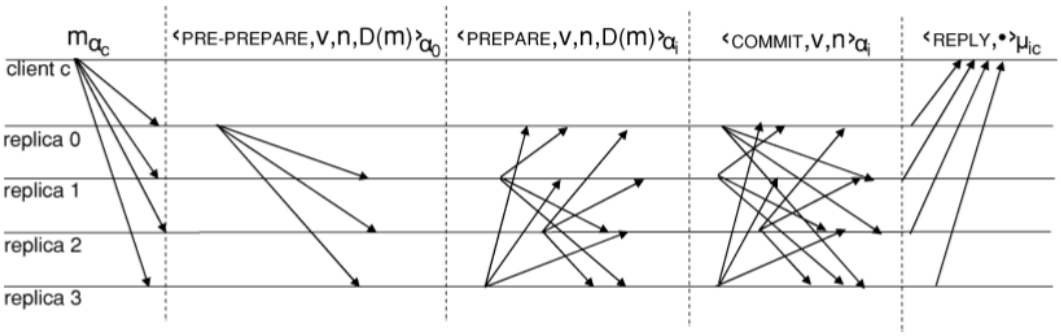

Fig. 1. Normal case operation: the primary (replica 0) assigns sequence number n to request m in its current view v and multicasts a PRE-PREPARE message with the assignment. If a backup agrees with the assignment, it multicasts a matching PREPARE message. When a replica receives messages that agree with the assignment from a quorum, it sends a COMMIT message. Replicas execute m after receiving COMMIT messages from a quorum.

每个副本的状态包括服务的状态，包含副本已接受或发送的消息的 message log，以及表示副本当前视图的整数。我们将在第 4.4 节中描述如何截断日志。状态可以保存在非永久性存储器中；它不需要稳定性。

当 primary p 从客户端接收到请求 m_{α_c}= <REQUEST, o, t, c>$_{\alpha_c}$时，如果它可以验证请求，则将序列号 n 分配给 m。然后，它将一个带有该分配的 PRE-PREPARE 消息多播到备份，并将该消息插入到其日志中。消息的形式为<PRE-PREPARE, v, n, D(m)>$_{\alpha_p}$，其中 v 表示消息将要发送的视图，D(m)是 m 的摘要。

像 PRE-PREPARE 一样，在其他阶段发送的 PREPARE 和 COMMIT 消息也包含 n 和 v。一个副本只接受以下消息之一，在视图 v 中的；可以验证消息真实性的；和 n 在低水位标记 h 和高水位标记 H 之间的。最后一个条件是必要的，以便能够进行垃圾收集并且通过选择非常大的序列号来防止一个故障 primary 耗尽序列号空间。我们在第 4.4 节中讨论 H 和 h 的进展情况。

备份 i 接受 PRE-PREPARE 消息，假如（除上述条件之外）它没有接受包含不同摘要的视图 v 和序列号 n 的 PRE-PREPARE。如果备份 i 接受 PRE-PREPARE，并且在其日志中具有请求 m，则通过多播一个⟨PREPARE, v, n, D(m), i⟩$_{\alpha_i}$消息和 m 的摘要到所有副本从而进入 prepare 阶段；此外，它将 PRE-PREPARE 和 PREPARE 消息都添加到其日志中。否则，它什么都不做。PREPARE 消息表示备份同意在视图 v 中分配序号 n 到 m。如果副本给一个请求发送了 PRE-PREPARE 或 PREPARE 消息，我们则认为这个请求在特定副本上是 pre-prepared 的。

然后每个副本收集消息，直到它具有带有 PRE-PREPARE 的 quorum 证书和 2 f 个与序列号 n，视图 v 和请求 m 匹配的 PREPARE 消息。我们称这个证书是 prepared certificate，我们称这个副本准备了（prepared）请求。该证书证明 quorum 已经同意在 v 中分配号码 n 到 m。该协议保证获得相同视图和序列号以及不同请求的 prepared certificate 是不可能的。

有趣的是，之所以这是正确的原因是因为它说明了 quorum certificates 的一种用途。假设它是假的，并且存在两个不同的，具有相同视图 v 和序列号 n 的 prepared certificate 的请求 m 和 m'。那么这些证书的 quorums 将至少有一个共同的非故障副本。该副本将发送 PRE-PREPARE 或 PREPARE 消息，同意在同一视图中为 m 和 m'分配相同的序列号。因此，m 和 m'不会有区别，这与我们的假设相违背。

这样可以确保副本对同一视图中的请求的总顺序达成一致，但是不能确保跨视图更改的

请求的总顺序。副本可以使用相同的序列号和不同的请求在不同的视图中收集 prepared certificates。提交阶段解决了以下问题。每个副本 i 多播$\langle \text{COMMIT}, v, n, i \rangle_{\alpha i}$称它有 prepared certificates，并将此消息添加到其日志。然后每个副本收集消息，直到它具有不同副本（包括其自身）中的相同序列号 n 和视图 v 的 2f + 1 个 COMMIT 消息的 quorum certificates。我们将此证书称为 committed certificate，并且称在该副本具有 prepared 和 committed certificates 时，该请求由副本提交。

在请求被提交之后，该协议保证该请求已经由 quorum 准备；也就是说，有一个知道 quorum 已接受了在视图 v 中分配号码 n 给一个请求的 quorum。新的 primaries 通过从 quorum 读取 prepared certificates 并在最新的视图的证书中选择序列号分配来确保关于已提交的请求的信息被传播到新的视图中。视图更改协议在第 4.5 节中有详细描述。

每个副本 i 执行在 m 提交时而且该副本执行了所有具有较低序列号的请求的客户端请求的操作。这样可以确保所有非故障副本按照提供安全性所需的相同顺序执行请求。执行请求的操作后，副本会向客户端发送回复。为了保证恰好一次语义，副本丢弃时间戳低于发送给客户端的最后一个答复中的时间戳的请求。

我们不依赖于有序的消息传递，因此副本可能提交有故障的请求。这并不重要，因为它保持 PRE-PREPARE，PREPARE 和 COMMIT 消息的记录，直到相应的请求可以执行。

如果客户端出现故障或请求在网络中损坏，则请求的认证者可能会同时有正确和不正确的 MACs。因此，有必要设计协议以确保副本同意请求是否可信。否则，这个问题可能导致违背安全性和活性。BFT 通过概括用于验证请求的真实性的机制来解决这个问题；如果请求的验证器中的副本 i 的 MAC 是正确的，或者 i 有具有在其日志中的请求摘要的 f + 1 个 PRE-PREPARE 或 PREPARE 消息，则 i 可以验证一个请求。第一个条件通常是足够的，但如果具有部分正确认证器的请求在某个正确的副本上提交，则第二个条件可以防止系统出现死锁。

4.4. Garbage Collection

本节讨论防止消息日志无限制增长的垃圾回收机制。副本必须从其日志中丢弃已经执行的请求的信息。但是，当副本执行相应的请求时，副本不能简单地丢弃消息，因为它可以丢弃稍后需要的 prepared certificate 以确保安全。相反，副本必须首先获得其状态正确的证明。然后，它可以丢弃与执行在该状态下反映的请求相对应的消息。

在执行每个操作之后生成这些证明将是昂贵的。相反，当执行具有由检查点周期（checkpoint period）K（例如，K = 128）可分割的序列号的请求时，它们可以周期性地生成。我们将执行这些请求所产生的状态称为检查点（checkpoints），我们认为带有证明的检查点是一个稳定的检查点（stable checkpoint）。

当副本 i 产生或提取检查点时，它将$\langle \text{CHECKPOINT}, n, d, i \rangle_{\alpha i}$消息多播到其他副本，其中 n 是执行在状态中反映的最后一个请求的序列号，d 是状态的摘要。副本维护服务状态的几个逻辑拷贝：最后一个稳定检查点，不稳定的零个或多个检查点以及当前状态。确保副本具有其稳定检查点的状态和匹配证明是必要的。第 6.2 节描述了如何有效地管理检查点和副本之间的传输状态。

每个副本收集消息，直到它具有由具有相同序列号 n 和摘要 d 的不同副本进行身份验证的 2f + 1 个 CHECKPOINT 消息（包括其自身）的 quorum certificate。我们将此证书称为稳定

证书（stable certificate）；它确保其他副本将能够获得一个弱证书，证明如果它们需要抓取它，则该稳定检查点是正确的。此时，序列号为 n 的检查点稳定，副本将丢弃其日志中所有具有小于等于 n 的序列号的条目；它也会丢弃所有较早的检查点。

检查点协议用于推进低 / 高水位标记（将何种消息添加到日志的限制）。低水位标记 h 等于最后稳定检查点的序列号，高水位标记为 H = h + L，其中 L 为日志大小。日志大小是副本将记录信息的连续序列号的最大值。通过将 K 乘以足够大的小常数因子（例如 2）获得，使得副本不可能停止等待检查点变得稳定。

4.5. View Changes

视图更改协议通过允许系统在 primary 发生故障时取得进展而提供活性。该协议还必须保护安全性：它必须确保非故障副本在视图中所提交的请求的序列号达成一致。

协议背后的基本思想是新的 primary 从 quorum 读取有关稳定和准备证书的信息，并将此信息传播到新视图。由于任何两个 quorum 相交，因此保证 primary 获取对先前视图中提交的所有请求以及所有稳定检查点的信息。本节的其余部分描述了可能需要无限空间的简化视图更改协议。我们对 Castro [2001]的消除了这个问题的协议进行了修改。

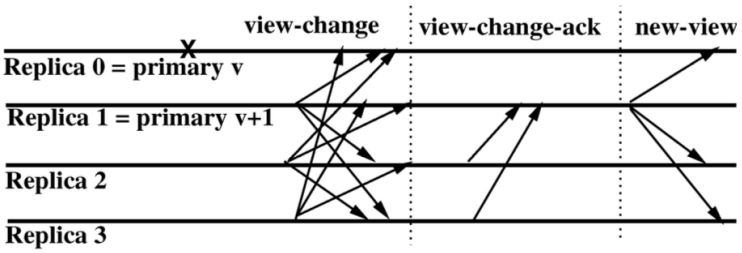

Fig. 2. View-change protocol: the primary for view v (replica 0) fails causing a view change to view $v+1$.

数据结构（Data Structures）

副本记录了早期视图中发生的情况。该信息保存在 P 和 Q 两个集合中。这些集合仅包含日志中当前高低水位线之间序列号的信息。即使在系统能够继续正常操作之前发生多个视图更改，这些集合也允许视图更改协议正常工作；当系统正常运行时，这些集合为空。副本还存储与这些集合中的条目相对应的请求。

在副本 i 处的 P 存储有关在以前的视图中在 i 上已经准备的请求的信息。它的条目是元组⟨n, d, v⟩，这意味着 i 在视图 v 中为一个具有号码 n 的摘要 d 的请求收集了一个 prepared certificate，并且在后面的具有相同号码的视图中没有在 i 上准备好的请求。

Q 存储关于在先前视图中已经在 i 上预准备（pre-prepared）的请求的信息（即，i 已经发送了 PRE-PREPARE 或 PREPARE 消息的请求）。它的条目是元组⟨n, d, v⟩，这意味着 i 在视图 v 中预准备了一个具有号码 n 的摘要 d 的请求，并且在后面的具有相同号码的视图中，请求没有在 i 上预准备。

View-Change Messages

图 2 展示了视图 v 到视图 v + 1 的视图变更协议。当备份 i 怀疑视图 v 的 primary 有故障

时，进入视图 v + 1，并多播一条⟨VIEW-CHANGE, v + 1, h, C, P, Q, i⟩$_{\alpha i}$ 消息给所有的副本。这里 h 是 i 已知的最新稳定检查点的序列号，C 是具有存储在 i 处的每个检查点的序列号和摘要的一组对，P 和 Q 是之前提到的集合。在使用日志中的信息发送 VIEW-CHANGE 消息之前，会更新这些集合，如图 3 所示。一旦发送了 VIEW-CHANGE 消息，i 将从其日志中删除 PRE-PREPARE，PREPARE 和 COMMIT 消息。如果算法重复改变视图却没取得进展，则 Q 中的元组数量可能无限制地增长。在 Castro [2001]中，我们描述了通过常数界定 Q 的大小的算法的修改。有趣的是，VIEW-CHANGE 消息不包括 PRE-PREPARE，PREPARE 或 CHECKPOINT 消息。

View-Change-Ack Messages

副本收集 V + 1 的 VIEW-CHANGE 消息，并将确认发送给 v + 1 的 primary，p。如果其 P 和 Q 组件中的所有信息都是小于或等于 v 的视图号，则副本只能接受这些 VIEW-CHANGE 消息。确认消息具有⟨VIEW-CHANGE-ACK, v + 1, i, j, d ⟩$_{\mu ip}$ 的形式，其中 i 是发送方的标识符，d 是正在确认的 VIEW-CHANGE 消息的摘要，j 是发送该 VIEW-CHANGE 消息的副本。这些确认消息允许 primary 证明由错误的副本发送的 VIEW-CHANGE 消息的真实性。

let v be the view before the view change, L be the size of the log, and h be the log's low water mark.

for all n such that $h < n \leq h + L$ do
  if request number n with digest d is prepared in view v then
    if $\exists \langle n, d', v' \rangle \in \mathcal{P}$
      remove $\langle n, d', v' \rangle$ from $\mathcal{P}$
    add $\langle n, d, v \rangle$ to $\mathcal{P}$
  if request number n with digest d is pre-prepared in view v then
    if $\exists \langle n, d, v' \rangle \in \mathcal{Q}$
      remove $\langle n, d, v' \rangle$ from $\mathcal{Q}$
    add $\langle n, d, v \rangle$ to $\mathcal{Q}$

Fig. 3. Computing $\mathcal{P}$ and $\mathcal{Q}$.

New-View Message Construction

新的 primary p 收集 VIEW-CHANGE 和 VIEW-CHANGE-ACK 消息（包括其自身的消息）。它将 VIEW-CHANGE 消息存储在集合 S 中。在从其他副本接收到的 i 的 VIEW-CHANGE-ACK 消息的 2f − 1 个 VIEW-CHANGE-ACK 之后，它将从副本 i 接收到的 VIEW-CHANGE 消息添加到 S。这些 VIEW-CHANGE-ACK 消息连同其接收到的 VIEW-CHANGE 消息以及它本可以发送的 VIEW-CHANGE-ACK 形成一个 quorum certificate。我们将其称为视图更改证书（view-change certificate）。S 中的每个条目都是用于不同的副本。

新的 primary 使用 S 中的信息和图 4 中绘制的决策过程来选择检查点和一组请求。该过程在每次 primary 接收到新信息时运行，例如，当它向 S 添加新消息时。我们使用符号 m.x 来指示消息 m 的组件 x，其中 x 是我们在定义 m 的消息类型的格式时使用的名称。

Primary 通过在新视图中选择要作为请求处理的起始状态的检查点来启动。它从一组已知正确的（因为它们具有弱证书），和号码高于至少 f + 1 个非故障副本的日志中的低水位线的检查点选择具有最高号码 h 的检查点。最后一个条件是活性的必要条件；它确保具有比 h 高定号码的提交的请求排序信息仍然可用。

接下来，primary 选择在 h 和 h + L 之间的每个序列号 n 的新视图中预准备的请求（其中 L 是日志的大小）。如果在以前的视图中提交了一个请求 m，则 primary 必须选择 m。如果存

在这样的请求，则它保证是唯一满足条件 A1 和 A2 的请求。条件 A1 确保 primary 选择在一个 quorum 中的几个副本声称已经在最新视图 v 中准备好了的请求，A2 确保请求可以在视图 v 中准备，因为它是由 v 中或后来的视图中至少一个正确的副本预准备的。

$$\text{let } \mathcal{C} = \{\langle n, d \rangle \mid \exists S, S' \subseteq \mathcal{S} : |S| > 2f \wedge |S'| > f \wedge \forall m \in S : m.h \leq n \wedge \forall m \in S' : \langle n, d \rangle \in m.\mathcal{C}\}$$

if $\exists \langle h, d \rangle \in \mathcal{C} : \forall \langle n', d' \rangle \in \mathcal{C} : n' \leq h$ then
select checkpoint with digest d and number h
else exit

for all n such that $h < n \leq h + L$ do
A. if $\exists m \in \mathcal{S}$ with $\langle n, d, v \rangle \in m.\mathcal{P}$ that verifies:
A1. $\exists 2f + 1$ messages $m' \in \mathcal{S}$:
$m'.h < n \wedge \forall \langle n, d', v' \rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$
A2. $\exists f + 1$ messages $m' \in \mathcal{S} : \exists \langle n, d', v' \rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$
then select the request with digest d for number n

B. else if $\exists 2f + 1$ messages $m \in \mathcal{S}$ such that $m.h < n \wedge m.\mathcal{P}$ has no entry for n
then select the null request for number n

Fig. 4. Decision procedure at the primary.

如果存在没有准备序列号为 n（条件 B）的任何请求的副本的 quorum，则不会有请求使用号码 n 来提交。因此，primary 选择一个通过协议作为常规请求但执行为无操作的特殊空请求。（Paxos [Lamport 1989]使用了类似的技术来弥补差距。）

当 primary 选择了每个号码的请求时，决策过程停止。这可能需要等待超过 n - f 个消息，但一旦 primary 接收到非故障副本为其视图发送的所有 VIEW-CHANGE 消息，则 primary 始终能够完成该决策过程。决定后，primary 通过其决策向其他副本多播一条 NEW-VIEW 消息：⟨NEW-VIEW, v + 1, V, X⟩$_{\alpha p}$。这里，V 包含由发送副本的标识符和其 VIEW-CHANGE 消息的摘要组成的 S 中的每个条目对，X 标识所选择的检查点和请求值。V 中的 VIEW-CHANGE 是 NEW-VIEW certificate。

New-View Message Processing

Primary 更新其状态以反映 NEW-VIEW 消息中的信息。它获取 X 中缺少的任何请求，如果它没有带序列号 h 的检查点，它也会启动协议以获取缺少的状态（参见第 6.2.2 节）。当它在 X 中具有所有请求，并且具有序列号 h 的检查点是稳定的时候，它在其日志中记录请求在视图 v + 1 中是预准备的。

视图 v + 1 的备份收集消息直到它们具有正确的 NEW-VIEW 消息和 V 中每个对的正确匹配的 VIEW-CHANGE 消息。如果备份没有收到 V 中某对副本的 VIEW-CHANGE 消息中的一个，primary 自己可能无法证明其收到的消息是真实的，因为它是没有签名的。使用 VIEW-CHANGE-ACK 消息解决了这个问题。由于 primary 在获得匹配的 view-change certificate 后，只包含 S 中的一条 VIEW-CHANGE 消息，所以至少 f + 1 个非故障副本可以保证其摘要在 V 中的每个 VIEW-CHANGE 消息的真实性。因此，如果原始 VIEW-CHANGE 的发送者不合作，则 primary 重传发送方的 VIEW-CHANGE 消息，非故障备份重传其 VIEW-CHANGE-ACK。如果一个备份接收到与 V 中的摘要和标识符相匹配的 f 个 VIEW-CHANGE-ACK，那么它则可以接受一条它的认证器是不正确的 VIEW-CHANGE 消息。

在获得 NEW-VIEW 消息和匹配的 VIEW-CHANGE 消息之后，备份通过执行图 4 中的决策过程来检查这些消息是否支持 primary 报告的决定。如果不是的话，则副本立即移动到视图 v + 2。否则，它们以与 primary 类似的方式修改其状态来对新信息负责。唯一的区别是，他们为每个标记为预准备的请求多播一条用于 v + 1 的 PREPARE 消息。此后，正常情况恢复运行。

4.5.1 正确性（Correctness）。我们现在非正式地认为，视图变更协议（view-change protocol）保留安全性（safety），而且它是活跃的（live）。

安全性（Safety）。我们首先草拟以下声明的证明。

如果请求 m 在视图 v 中的某个正确的副本上提交序列号 n，则没有其他请求在另一个正确副本中提交 v 和 n，并且图 4 中的决策过程将不会在任何 v' > v 的视图中选择对序列号 n 的不同请求。

这个声明意味着在请求中提交序列号为 n 的视图 v 中，没有明确的请求可以在具有与 v 相同的视图的相同序列号的任何正确副本上进行预准备。因此，正确的副本对请求的总顺序达成一致，因为它们从不提交具有相同序列号的不同请求。

证明是通过对 v 和 v'之间的视图号码进行归纳。如果 m 在一些正确的副本 i 上提交了，则 i 从副本 Q 的 quorum 接收了 COMMIT 消息，就可以说他们准备了具有序列号 n 和视图 v 的请求。通过 quorum intersection 属性，不同的请求不能在具有相同的视图和序列号的正确的副本上准备。因此，在 v' = v 的基础情况下，这种说法是正确的。

对于归纳步骤（v' > v），假设一个矛盾，决策过程在 v'中为序列号 n 选择一个请求 m' ≠ m。这意味着不管条件 A1 还是条件 B 都必须为真。通过 quorum intersection 属性，必须至少有一个来自具有在用来满足条件 A1 和条件 B 的任意 quorum certificate 中的 h < n 对正确副本 j ∈ Q 的 VIEW-CHANGE 消息。

从归纳假设和图 3 所述的计算 P 的过程中，j 中 v'的 VIEW-CHANGE 消息必须在它的 P 组件 vc ≥ v 中包括⟨n, D(m), v_c⟩（因为 j 没有为序列号 n 的信息进行垃圾回收）。因此条件 B 不可能为真。但是如果来自错误的副本的 VIEW-CHANGE 消息在其 $v_f > v_c$ 的 P 组件中包括<n, D(m'), v_f>，条件 A1 可以为真；条件 A2 可以防止这个问题。条件 A2 只有在其 Q 组件中具有<n, D(m'), v_c'>的正确副本有 VIEW-CHANGE 消息时才为真，使得 $v_c' \geq v_f$。由于 D(m') ≠ D(m)（概率很高），归纳假设意味着 $v_c' \leq v$。因此，$v_f \leq v$ 以及条件 A1 和 A2 不能都为真，即完成了证明。

活性（Liveness）。为了提供活性，如果副本无法执行请求，则必须移动到新视图。视图变更由阻止备份无限期等待执行请求的超时或者当备份检测到 primary 有故障的时候触发。如果一个备份收到一个有效的请求并且没有执行，则备份正在等待请求。备份在接收到请求并且定时器尚未运行时启动定时器。当它不再等待执行请求时，它会停止定时器，但是如果在那个时候它正在等待执行其他请求，它将重新启动定时器。

我们现在可以非正式地认定，该算法是具有活性的。我们首先声明，一个正确的 primary 将能够发送一个 NEW-VIEW 消息，前提是它有足够的时间使得正确的副本转到下一个视图。然后我们解释一下算法如何最大限度地提高完成视图更改的可用时间并处理一些新的请求。

反证假设，具有无限时间的正确 primary 无法使用图 4 中的过程进行决策。我们从表明至少有一个满足决策过程中的条件的检查点开始。通过选择此检查点或满足这些条件的任何

其他检查点，使得 primary 能够取得进展。用 h_c 表示在某些正确副本上稳定的最新检查点的序列号。由于至少有 2f + 1 个正确的副本，并且至少 f + 1 个正确的副本具有序列号为 hc 的检查点，所以 primary 将能够为 h 选择值 h_c。如果需要进行进展，副本将能够获取 primary 选择的任何检查点，因为至少有一个正确的副本具有检查点。

对于 h 和 h + L 之间的每个序列号 n，我们认为 primary 可以选择一个满足条件 A 或 B 的请求。情况包括：（1）一些正确的副本准备了一个序列号为 n 的请求；或者（2）没有这样的副本存在。在情况（1）中，条件 A1 将被验证，因为有 2f + 1 个非故障副本，并且非故障副本从不为同一视图和序列号准备不同的请求；A2 也将得到满足，因为在非故障副本上准备的请求会预准备至少 f + 1 个非故障副本。此外，条件 A2 意味着至少有一个有正确副本的请求证明其真实性。因此，缺少所选请求的任何副本都可以获取它，并且可以相信它是真实的。在情况（2）中，条件 B 最终将被满足，因为有 2f + 1 个正确的副本，假设没有准备任何序列号 n 的请求。

最大化在至少 2 f + 1 个非故障副本处于相同视图中并且其中一个是 primary 的时候这段时间是非常重要的。此外，我们可以调整超时，以确保这段时间指数增加，直到某些操作执行。我们通过以下几种手段实现这些目标。

首先，为了避免启动视图变更太快，给视图 v + 1 多播 VIEW-CHANGE 消息的副本将在启动定时器之前等待给视图 v + 1 的 2f + 1 个 VIEW-CHANGE 消息。然后，它会在一段时间 T 之后启动其定时器直至过期。如果定时器在收到 v + 1 的有效 NEW-VIEW 消息之前或在新视图中执行以前未执行的请求之前到期，则它将为视图 v + 2 启动视图变更，但这次它将等待 2T，然后开始视图 v + 3 的视图变更。

其次，如果一个副本收到来自大于其当前视图的视图的其他副本的一组 f + 1 个有效的 VIEW-CHANGE 消息，则它会为该集合中的最小视图发送一个 VIEW-CHANGE 消息，即使其定时器还未过期；这样可以防止下一次视图变更启动地太晚。

第三，故障副本无法通过强迫频繁的视图变更来阻碍进展。一个故障副本不能通过发送 VIEW-CHANGE 消息来导致视图变更，因为仅当至少 f + 1 个副本发送 VIEW-CHANGE 消息时才会发生视图变更。但是当它是 primary 的时候（通过不发送消息或发送不良消息），可以导致视图变更。然而，由于视图 v 的 primary 是副本 p，使得 p = v mod | R |，primary 不可能超过 f 个连续视图是故障的。

这三种技术提供了活性，除非消息延迟比无限期的超时增长地更快，而这在实际的系统中是不太可能的。

我们的实现保证了公平性：它确保客户端即使在其他客户端访问服务时也能够对其请求进行回复。一个无故障的 primary 使用 FIFO 规则分配序列号。备份在 FIFO 队列中维护请求，并且只有执行队列中的第一个请求时才停止视图更改定时器；这样可以防止有故障的 primary 在不处理来自其他客户端的请求时给予某些客户端优先权。

5. BFT-PR: BFT WITH PROACTIVE RECOVERY

如果在系统生命周期内少于 1/3 的副本失败，则 BFT 提供安全性和活性。这些保证对于长期存在的系统是不够的，因为在这种情况下可能会超出限制。因此，我们为再次产生故障副本行为的 BFT 开发了恢复机制。具有恢复功能的 BFT，BFT-PR，可以容忍任何数量的故障，

倘若少于 1/3 的副本在短期漏洞窗口中出现故障。

一个拜占庭故障副本可能在被破除时表现正常；因此，恢复必须是主动的，以防止攻击者在没有被检测到的情况下损坏 1/3 的副本而损害服务器。即使没有理由怀疑它们有缺陷，我们的机制也会定期恢复副本。

第 5.1 节描述了提供自动恢复所需的额外假设，第 5.2 节介绍了修改后的算法。

5.1. Additional Assumption

为了实现恢复，我们必须共同验证一个恢复到其它副本的故障副本，并且我们需要一个可靠的机制来触发定期的恢复。这可以通过让系统管理员参与恢复过程来实现，但是鉴于我们经常恢复副本以实现短期漏洞窗口的目标，这种方法是不切实际的。为了实现自动恢复，我们需要额外的假设。

安全加密（Secure Cryptography）。每个副本都有一个安全的加密协处理器，例如达拉斯半导体iButton或IBM PC 300PL主板的安全芯片。协处理器存储副本的私钥，并且可以在不暴露此密钥的情况下对消息进行签名和解密。它还包含一个不会倒退的计数器。这使它能够将计数器附加到它签署的消息中。

只读内存（Read-Only Memory）。每个副本将其他副本的公钥存储在某些内存中，而不会被破坏。该内存可能是 Flash BIOS 的一部分。大多数主板都可以进行配置，以便在需要时有机器的物理访问权限来修改 BIOS。

看门狗定时器（Watchdog Timer）。每个副本都有一个看门狗定时器，它将周期地中断处理和手动控制一个存储在只读内存中的恢复监视器。为了使这种机制有效，攻击者不应该在没有机器的物理访问权限的情况下更改看门狗中断的速率。有扩展卡来提供此功能。

当攻击者没有副本的物理访问权限时，这些假设可能会被挂起，我们期望这是常见的情况。当它们失败时，我们可以在系统管理员上回退来执行恢复。

注意，之前所有的主动安全算法（[Ostrovsky and Yung 1991; Herzberg et al. 1995, 1997; Canetti et al. 1997; Garay et al. 2000]）都假设由副本运行的整个程序是在只读内存中的，因此它不能被攻击者修改，并且大多数还假定即使在副本从危害中恢复后仍然继续工作的副本之间是有认证通道的。这些假设将足以实现我们的算法，但它们在实践中不太可能成立。我们只需要在只读内存中使用小型监视器，并使用安全协处理器在恢复后在副本之间建立新的会话密钥。

唯一没有采用认证通道的主动安全性的成果是 Canetti et al. [1997]，但是副本可以在其私钥泄密时能够做到的最好的办法是提醒管理员。我们的安全加密假设可以使得从大多数故障中自动恢复，并且现在可以使用我们所需的属性来保护协处理器。我们还假定客户端有一个安全的协处理器；这简化了客户端和副本之间的密钥交换协议但是可以通过向该协议添加一个额外的回合来避免这种情况。当目标是容忍不是由恶意智能引发的故障时，可以放宽这些假设。

带有主动恢复的 BFT 需要更强的同步假设来提供活性。我们假定在所有消息都在某个常量时间 delta 内(可能在重新传输之后)或所有非故障客户端都收到了对请求的回复的时候之后的执行过程中存在一些未知的点。这里 delta 是一个常数，它取决于算法使用的超时值。

这个假设比迄今为止使用的假设更强，以允许以固定速率进行恢复，但它仍然可能在有一个适当的 delta 的选择的实际系统中保持。

5.2. Modified Algorithm

我们首先概述恢复机制。然后我们详细解释它。

5.2.1 概述。BFT 使用 quorum 作为可靠内存来存储请求排序信息。我们必须确保这种内存在主动恢复的情况下保持工作。特别是主动恢复机制必须确保以下内容。

非故障副本收到的每个 quorum certificate 必须被一个 quorum 支持；也就是说，非故障 quorum 成员的状态必须记录一个匹配消息被发送，或者他们必须有一个更新的稳定的检查点。

另外，恢复机制必须确保副本所持的服务器状态与协议状态一致：

对于任何非故障复制品，具有序列号 n 的当前服务器状态（或任何检查点）的值必须与通过运行具有 h + 1 和 n 之间递增顺序的序列号并且由稳定检查点 h 启动的请求而获得的值相同。这些请求必须在副本上提交。

有几个问题需要解决，以确保在副本恢复时保留这些不变性。首先，有必要防止攻击者在恢复后冒充副本。否则，没有希望确保上述任何不变量。如果攻击者掌握用于验证消息的 MAC 密钥，即使使用安全加密协处理器签名消息，也可能会发生伪造情况，攻击者可以在控制故障副本时签署不良消息。我们通过在恢复期间更改 MAC 密钥以及让副本和客户端拒绝使用旧密钥进行身份验证的消息来避免此问题。

但是，只更改密钥是不够的。如果副本在足够长的时间内收集证书的消息，则当副本出现故障时，可能会收到由副本发送的多于 f 个的消息而终止，这违反了第一个不变性。我们通过让副本和客户端在更改密钥时丢弃不是完整证书的一部分的所有消息来解决此问题。为了确保活性，副本和客户端使用最新的密钥验证他们重新发送的消息。第 5.2.2 节解释了密钥如何改变。

由于恢复是主动的，恢复的副本可能不会有问题，恢复必定不会导致其出现故障；否则上述任何不变性都可能被违反。特别地，一个非故障副本不会丢失其状态，并且我们需要允许它在恢复时继续参与请求处理协议，因为有时在完成恢复时是需要的。然而，如果恢复的副本实际上是故障的，则恢复机制必须确保其状态达到满足上述不变性的值，并且必须防止副本传播不正确的信息。其困难在于恢复期间我们不知道恢复的副本是否有故障。我们将在 5.2.3 节中解释如何解决这个问题。

5.2.2 密钥交换（Key Exchange）。副本和客户端通过定期（例如每分钟）发送 NEW-KEY 消息来刷新用于验证发送给他们的消息的会话密钥。相同的机制被用来建立初始会话密钥。消息具有$\langle$NEW-KEY, I, ...，$\{k_{j,i}\}_{\in j}$, ..., t$\rangle_{\sigma i}$ 的形式。消息由安全协处理器（使用副本的私钥）签名，t 是其计数器的值；计数器由协处理器递增，并在每次生成签名时附加到消息中。（这防止了抑制重放攻击[Gong 1992]。）每个 $k_{j,i}$ 是 j 应该用于验证将来发送给 i 的消息的密钥副本；$k_{j,i}$ 被 j 的公钥加密，所以只有 j 可以读取它。副本使用时间戳 t 来检测虚假的 NEW-KEY 消息：t 必须大于从 i 接收的最后一个 NEW-KEY 消息的时间戳。

每个副本与每个客户端共享一个密钥；该密钥用于双向通信。密钥由客户端使用 NEW-KEY 消息定期刷新。如果一个客户端在某些系统定义的时间段内忽略了这一点，则每

个副本将丢弃该客户端的当前密钥，这会迫使客户端刷新密钥。

令 t_1 和 t_2（$> t_1$）是两个连续的 NEW-KEY 消息由同一个节点发送的时刻。我们将时间间隔[t1, t2]称为一个刷新时期，其持续时间，$t_2 - t_1$，为一个刷新周期。

当副本或客户端发送一个 NEW-KEY 消息时，它会丢弃其日志中不属于完整证书一部分的所有消息（其发送的 PRE-PREPARE 和 PREPARE 消息除外），并拒绝未来接收到的任何由旧密钥验证的消息。这确保正确的节点只接受具有同样新鲜的消息的证书，即使用在同一刷新周期创建的密钥进行身份验证的消息。

5.2.3 恢复（Recovery）。恢复协议使故障副本再次正常运行，以允许系统在其生命周期容忍多于 f 个故障。为了实现这一点，协议确保在副本恢复之后：它运行正确的代码，它不能被攻击者模仿，并且其状态满足之前定义的不变性。协议经过以下步骤。

重启（Reboot）。恢复是主动的—当看门狗定时器关闭时，它会定期启动。如果恢复的副本相信它是视图 v 的 primary，则在开始恢复之前多播一个 VIEW-CHANGE 消息到 v + 1。任何接收此消息并且在视图 v 中的正确的副本立即变更到视图 v + 1。这可以提高可用性，因为备份在更改为 v + 1 之前不必等待其计时器到期。一个故障的 primary 可能会发送此类消息并强制视图变更，但这不是问题，因为替换一个故障的 primary 总是没错的。

恢复监视器将副本的状态（日志，服务器状态和检查点）保存到磁盘。然后，使用正确的代码重新启动系统，并从保存的状态重新启动副本。可以通过将其摘要存储在只读内存中并使恢复监视器检查此摘要来确保操作系统和服务器代码的正确性。如果副本存储的代码的拷贝已损坏，则恢复监视器可以从其他副本获取正确的代码。或者，整个代码可以存储在只读介质中；这是可行的，因为有几个可以通过物理上关闭跳线开关（例如，希捷猎豹 18LP）来保护地写入的磁盘。重新启动将操作系统数据结构恢复到正确的状态，并删除攻击者留下的任何特洛伊木马。

此后，恢复的副本的代码是正确的，并没有失去其状态。副本必须保留其状态，并使用它来处理请求，即使它正在恢复。当恢复的副本没有错误时，这对于确保常见情况下的安全性和活性至关重要；否则恢复可能导致第 f + 1 个故障。但是，如果恢复的副本有故障，则状态可能已损坏，并且攻击者可能会伪造消息，因为它知道用于对传入和传出消息进行身份验证的 MAC 密钥。恢复协议可以如下所述地解决这些问题。

恢复副本 i 通过丢弃与客户端共享的密钥并且多播一条 NEW-KEY 消息以更改其用于认证由其他副本发送的消息的密钥开始。这一点很重要，因为不然的话，如果 i 存在故障，攻击者可以通过模拟任何客户端或副本来阻止一次成功的恢复。

运行评估协议（Run Estimation Protocol）。接下来，i 运行一个简单的协议来评估高水位的上限 H_M，如果它没有故障，它将出现在其日志中；它会丢弃具有更大序列号的任何日志条目或检查点。这限制了副本发送的任何不正确的消息的序列号，同时确保当副本没有故障时，不会丢弃任何状态。

评估工作如下：i 向其他副本多播一个$\langle$QUERY-STABLE, i$\rangle_{\alpha i}$ 消息。当副本 j 接收到该消息时，它回复$\langle$REPLY-STABLE, c, p, i$\rangle_{\mu ji}$，其中 c 和 p 分别是最后一个检查点和在 j 准备的最后一个请求的序列号。副本 i 不断重新发送查询消息并处理回复；它保持 c 的最小值和从每个副本接收的 p 的最大值。它还保留了自己的 c 和 p 值。在评估期间，除了 NEW-KEY 和 REPLY-STABLE 之外，i 不处理任何其他协议消息。

恢复的副本使用响应选择 H_M 如下。$H_M = L + c_M$，其中 L 是对数大小，c_M 是从满足两个条件的一个副本 j 接收的 c 值，两个条件是：除了小于或等于 c_M 的 c 的 j 上报值以外的 2f 个副本，以及除了大于或等于 c_M 的 p 的 j 上报值以外的 f 个副本。

为了安全性，c_M 必须大于任何稳定检查点 i 无故障时可能会有的序列号，以便在这种情况下不会丢弃日志条目。这是确定的，因为如果检查点稳定，则它将至少由 f + 1 个非故障副本创建，并且它的序列号小于或等于它们提出的 c 的任何值。对 p 的测试确保 c_M 靠近某些非故障副本的检查点，因为至少一个非故障副本报告 p 不小于 c_M；这很重要，因为它可以防止故障副本延长 i 的恢复。评估是具有活性的，因为有 2f + 1 个非故障副本，而且如果相应的请求被提交，他们只提出 c 的一个值；这意味着它至少准备了 f + 1 个正确的副本。因此，i 总是可以以正确的副本发送的消息集上的 cM 的选择为基础。

在这一点之后，i 就好像没有恢复一样参与到协议中，但它不会发送任何具有高于 H_M 的序列号的消息，直到其具有大于或等于 H_M 的序列号的正确的稳定检查点。这可以确保根据损坏状态可能发送的任何不良消息的序列号的绑定 H_M。

发送恢复请求（Send Recovery Request）。接下来，i 将恢复请求多播到其他副本，形式如下：$\langle \text{REQUEST}, \langle \text{RECOVERY}, H_M \rangle, t, i \rangle_{\sigma i}$。该消息由加密协处理器生成，t 是协处理器的计数器，以防止重播攻击。如果它是重播，或者如果他们最近接受了 i 的恢复请求（最近可以定义为看门狗时期的一半），则其他副本拒绝该请求。这对于防止在非故障副本正在忙着执行恢复请求时发生 denial-of-service 攻击非常重要。

恢复请求被和任何其他请求同样对待：它被分配一个序列号 n_R，并且经历通常要经历的三个阶段。但是当另一个副本执行恢复请求时，它会发送自己的 NEW-KEY 消息。副本还在获取缺省状态时发送一条 NEW-KEY 消息（参见第 6.2.2 节），并确定它反射了新的恢复请求的执行。这很重要，因为如果恢复的副本有故障，攻击者可能会知道这些密钥。通过更改这些密钥，我们限制了攻击者伪造可能被其他副本接受的消息的序列号—这保证它们在恢复请求执行时，不会接受具有大于日志中最大高水位的序列号的伪造消息；即 $H_R = \lfloor n_R / K \rfloor \times K + L$。

对恢复请求的回复包括序列号 n_R。副本 i 使用与客户端相同的协议来收集对其恢复请求的正确答复，但要等待 2 f + 1 个回复。然后计算其恢复点 $H = \max(H_M, H_R)$。副本还计算一个有效的视图：如果有 f + 1 个大于或等于 v_r 的视图的恢复请求的回复，则保留其当前视图 v_r，否则它将更改为回复中的视图的中值。如果副本在恢复启动后变更到其视图，该副本还会保留该视图。如果副本更改其视图，则会为 v_m 发送 VIEW-CHANGE 消息，并在 v_m 中生效之前等待正确的 NEW-VIEW 消息和匹配的一组 VIEW-CHANGE 消息。

计算有效视图的机制可确保非故障副本永远不会更改为号码小于上一个活跃视图的视图。如果恢复的副本是正确的并且具有号码为 v_r 的活跃视图，则有一个副本的 quorum，其视图号大于或等于 v_r。因此，恢复请求将不会在视图号小于 v_r 的任何正确的副本上准备。另外，在回复请求中的视图编号的中位数将大于或等于来自正确副本的回复中的视图编号。因此，它将大于或等于 v_r。在回复中更改为视图数的中值 v_m 也是安全的，因为至少一个正确的副本在视图数大于或等于 v_m 的情况下执行了恢复请求。

检查和获取状态（Check and Fetch State）。当 i 正在恢复时，它使用在第 6.2.3 节中讨论的状态转移机制来确定状态的哪些页面已损坏，并获取已过期或损坏的网页。

当副本 i 具有大于或等于 H 的序列号的稳定检查点时，它就会被恢复。如果客户端不使用系统，则可能会延迟恢复，因为请求号 H 需要执行恢复以完成。但是，这很容易解决。在发生恢复时，primary 为空请求发送 PRE-PREPARE 消息。

我们的协议具有不错的属性，任何副本都知道，当检查点 H 稳定并且已经收到来自 i 的 CHECKPOINT 消息时，i 已经完成了其恢复。这允许副本估计 i 恢复的持续时间，这对于检测减速具有低假阳性的恢复的 denial-of-service 攻击是有帮助的。

5.2.4 改进的服务属性（Improved Service Properties）。BFT-PR 确保一次执行 τ 的安全区和活性，倘若之多 f 个副本在 $T_v = 2T_k + T_r$ 的任何时间间隔内失败。这里，T_v 是漏洞窗口，T_k 是在非故障节点的 τ 中的最大密钥更新周期，T_r 是副本失败时和其在 τ 中从故障恢复时之间的最大时间。注意，T_k 和 T_r 的值是每个执行 τ 的特征，算法对其是未知的。

有必要将漏洞窗口设置为大于或等于 $2T_k + T_r$ 的值，以确保正确的节点不收集有超过 f 个不良消息的证书。用较小的窗口保留第 5.2.1 节中列出的不变性是没有什么希望的。会话密钥刷新机制确保非故障节点仅接受在大小最多为 $2T_k$ 的间隔内生成的消息的证书[1]。此外，限制在 $T + T_r$（对于任何 T）的间隔内可能失败的副本的数量可以确保在任何大小至多 T 的间隔内，永远不会有超过 f 个错误的副本。因此，正确节点收集的任何证书都包含由副本故障时发送的最多 f 个消息。

接下来我们认为恢复机制保留了第 5.2.1 节中列出的不变性。我们设计了恢复机制，以确保非故障副本在恢复时不会丢失其状态。因此，在这种情况下保留了不变性。当恢复的副本有故障时，不变性也会被保留下来。这是现实的，因为其他正确的副本不接受由序列号大于恢复点的恢复副本发送的不良消息。此外，副本具有正确的日志和正确的序列号等于恢复结束时的恢复点的稳定检查点。这确保副本具有一个稳定的检查点，其序列号大于在恢复期间或恢复之前被另一副本或客户端接受作为证书的一部分的发送的任何消息。

我们几乎无法控制 T_v 的值，因为 T_r 可能被一次 denial-of-service 攻击增加。但是我们对 T_k 和看门狗超时 T_w 之间的最大时间有很好的控制，因为它们的值由定时器速率决定，这是非常稳定的。设置这些超时值涉及安全性和性能之间的权衡：小的值通过减短漏洞窗口来提高安全性，但是引起更频繁的恢复和密钥变更以降低性能。第 8.2.3 节显示，这些超时可能相当小，导致性能下降非常低。

密钥变更之间的周期 T_k 可以很小，而不会对性能产生重大影响（例如，15 秒）。但在正常负载条件下，T_k 应大大超过三个消息延迟，以提供活性。

T_w 的值应根据 R_n 来设置，R_n 是在正常负载条件下恢复非故障副本所花费的时间。当以前的恢复尚未完成时，恢复副本是没有意义的；并且我们错开了恢复，以至于不超过 f 个副本立刻恢复，因为否则的话，即使没有攻击，服务也会中断。因此我们设置 $T_w = 4 \times s \times R_n$。这里的因子 4 代表了 3f +1 个副本 f 在同一时间的交错恢复，s 代表良性过载条件（即没有攻击）的安全性因子。图 5 显示了各种时间间隔之间的关系。

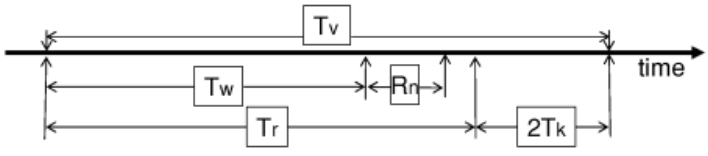

Fig. 5. Relationship between the window of vulnerability T_v and other time intervals.

第 8.2.3 节的结果表明，R_n 由用来重新启动和检查服务器状态的副本拷贝的正确性的时间主导。由于没有故障的副本检查其状态，而不会对网络或任何其他副本造成太大的负担，

[1] It would be T_k except that during view changes replicas may accept messages that are claimed authentic by f C 1 replicas without directly checking their authentication token.

我们预计并行恢复 f 副本的时间以及在良性过载条件下恢复副本的时间会接近 R_n；因此我们可以把 s 设置的接近 1。

我们不能保证在 denial-of-service 攻击下对 T_v 进行任何限制，但如果副本需要超过某个常数 R_n 的时间，对副本时间恢复并提醒管理员是可行的。然后，管理员可以采取措施来允许恢复终止。例如，如果副本通过私有网络连接，则它们可能会停止处理传入的请求并使用私有网络来完成恢复。这将中断服务器，直到恢复完成，但不会给攻击者带来任何好处；如果攻击者可以防止恢复完成，它也可以防止请求的执行。它可能会自动执行此响应。

副本还应记录关于恢复的信息，包括恢复节点是否存在故障，以及恢复需要多长时间，因为此信息对于加强系统以防未来的攻击是有帮助的。

6. IMPLEMENTATION TECHNIQUES

本节介绍协议优化和检查点管理。

6.1. Optimization

本节将介绍在正常情况下运行时提高性能的优化，同时保持安全性和活性。最重要的优化已经被介绍过了：BFT 使用基于对称加密的 MAC 来认证消息而不是公共密钥签名。由于 MAC 可以更快地计算三个数量级的排序，所以这种优化是非常有效的。

摘要回复（Digest Replies）。第二个优化在操作得打很大的结果时，显着降低了网络带宽消耗和 CPU 开销。一个客户端请求指定一个副本发送结果。该副本可以随机地或使用某种其他负载平衡方案来选择。在指定的副本执行请求之后，它会发送回来一个包含结果的回复。其他副本只发送包含结果摘要的回复。客户端收集至少 f + 1 个回复（包括具有结果的那个回复），并使用摘要来检查结果的正确性。如果客户端没有从指定的副本收到正确的结果，它则会重新发送请求（像往常一样）请求所有副本发送带有结果的回复。

暂行执行（Tentative Execution）。第三个优化将操作调用的消息延迟数从 5 减少到 4 个。一旦副本为请求准备好一个证书；它们的状态反射所有有较低序列号的请求的执行；以及这些请求已被提交，则这些副本立即临时地执行请求。执行请求后，副本会向客户端发送临时回复。由于回复是暂时的，客户端必须等待具有相同结果的答复的 quorum certificate。这确保了请求是由 quorum 准备的，因此，它被保证最终在非故障副本上提交。如果客户端的重传定时器在接收到这些回复之前到期，则客户端重新发送请求并等待非临时的回复。

暂时执行的请求可能会在视图变更时中止。在这种情况下，副本会将其状态恢复到 NEW-VIEW 消息中的检查点或其最后一个检查点状态（取决于哪一个具有较高的序列号）。

可以利用临时执行来消除 COMMIT 消息：它们可以在由副本发送的下一个 PRE-PREPARE 或 PREPARE 消息中搭载。由于客户端在请求准备后收到回复，因此，背负 COMMIT 不会增加延迟，而且它会减少网络和副本 CPU 上的负载。

只读操作（Read-Only Operations）。这种优化可以改善不修改服务器状态的只读操作的性能。客户端将只读请求多播到所有副本。副本在检查请求是否被正确认证后立即执行，客户端具有访问权限，并且该请求实际上是只读的。只有在只读请求提交之前所有请求执行之

后，副本才会发送回来一条回复。客户端等待具有相同结果的答复的 quorum certificate。如果对影响结果的数据进行并发写入，则可能无法收集到此证书。在这种情况下，在重发定时器超时之后，它将该请求作为常规读写请求重传。

只要客户端获得一个 quorum certificate，其回复不仅仅是只读操作，而且用于任何读写操作，只读优化就可以保留线性化。这种优化可以减少对大多数只读请求的单次往返的延迟。

请求批处理（Request Batching）。批处理通过为一批请求分配单个序列号并通过启动批次的协议的单个实例来减少负载下的协议开销。我们使用滑动窗口机制来限制可以并行运行的协议实例的数量。令 e 是 primary 执行的最后一批请求的序列号，令 p 是 primary 发送的最后一个 PRE-PREPARE 的序列号。当 primary 接收到一个请求时，它会立即启动协议，除非 $p \geq e + W$，其中 W 是窗口大小。在后一种情况下，它会排队请求。当请求执行时，窗口向前滑动以允许排队的请求进行处理。然后，primary 从队列中取出第一个请求，使得它们的大小之和低于常量边界，它为它们分配序列号，并将它们发送到单个 PRE-PREPARE 消息中。该协议的进行方式与单个请求完全相同，除了副本执行批次的请求（按照它们被添加到 PRE-PREPARE 消息的顺序）和它们为每个请求分别发送回去的回复。

6.2. Checkpoint Management

BFT 的垃圾收集机制（见第 4.4 节）采用称为检查点的服务器状态的逻辑快照。这些快照用于替换已经在日志中进行垃圾回收的消息。本节介绍一种管理检查点的技术。它由描述检查点创建，检查点摘要的计算和用于记录检查点信息的数据结构开始。然后，它描述了一种状态传输机制，用于将副本更新到它们丢失的某些消息被垃圾回收的时刻。它通过解释恢复过程中用于检查副本状态正确性的机制来结束。

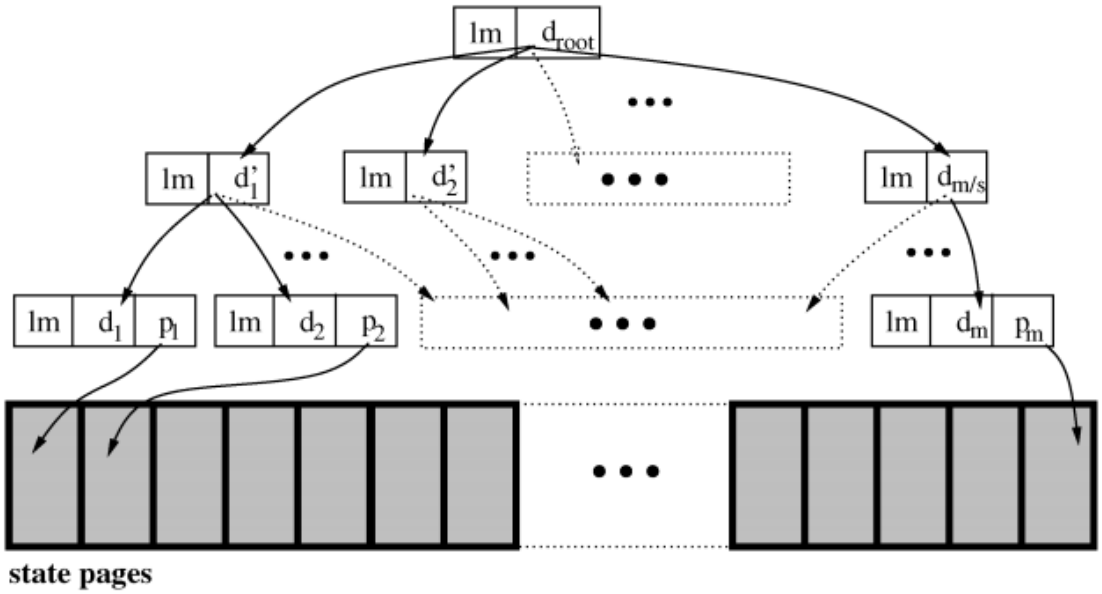

Fig. 6. Partition tree.

6.2.1 数据结构（Data Structure）。我们使用分层状态分区来降低计算检查点摘要和传输的信息量的成本以使副本最新。根分区对应于整个服务器状态，每个非叶分区被划分为 s 个相等大小的连续子分区。图 6 描绘了具有三个等级的分区树。我们把叶分区称为页面（pages），内部分区称为元数据。例如，第 8 节中描述的实验运行的层次结构具有四个级别，

s 等于 256，页面大小为 4-KB。

每个副本维护每个检查点的分区树的一个逻辑拷贝。当检查点被采集时，创建副本，并且当稍后的检查点变得稳定时，丢弃该副本。在临时执行序列号可以被检查点周期 K 整除的请求批次之后立即采集检查点（但是相应的 CHECKPOINT 消息仅在批处理提交之后提交）。

检查点的树为每个元数据分区存储一个元组⟨lm, d⟩，为每个页面存储一个元组⟨lm, d, p⟩。这里，lm 是分区被修改的最后一个检查点时期的结尾的检查点的序列号，d 是分区的摘要，p 是页面值。分区摘要很重要。副本在视图变更时期使用根分区的摘要来在没有传输大量数据的新视图中处理的请求的初始状态上达成一致。它们也用于减少在状态转移期间发送的数据量。

摘要如下被有效计算。通过将加密散列函数（当前为 MD5 [Rivest 1992]）应用于通过连接状态内的页面索引，其 lm 和 p 的值而获得的字符串来获得页面摘要。通过将散列函数应用于通过将其级别内的分区的索引（其 lm 值）与其子分组的摘要的大整数的模数之和获得的字符串来获得元数据摘要。因此，我们在每个元数据级别应用 AdHash [Bellare and Micciancio 1997]。这种结构的优点在于，通过从先前检查点逐步更新摘要，可以有效地获得检查点的摘要。它受到 Merkle 树的启发[Merkle 1987]。

分区树的拷贝是符合逻辑的，因为我们使用写时复制，以便只有从检查点获取以来修改的元组的拷贝才会被存储。这显著减少了维护这些检查点的空间和时间开销。

6.2.2 状态转移（State Transfer）。当一个副本发现一个序列号大于其日志中的高水位的稳定检查点时，副本会启动状态转移。它使用状态转移机制来获取对缺失的服务器状态的修改。副本可以通过接收 CHECKPOINT 消息或视图变更来发现这样一个检查点。

状态转移机制要有效率是非常重要的，因为它用于在恢复期间更新副本，而且我们经常要进行主动恢复。实现高效率的关键问题是减少传递的信息量，减少其他副本的负担。有效地获取状态的策略是向下递归分区层次结构以确定哪些分区已过期。这减少了需要提取的（非叶分区和叶分区）分区的信息量。

状态转移机制还必须确保即使某些副本有故障或状态被同时修改时，转换状态也是正确的。这个想法是这样的，分区的摘要提交了特定序列号的所有子分区的值。副本通过在某个检查点 c 获取具有根分区的摘要的弱证书来启动状态转移。然后它使用这个摘要来验证它获取的子分区的正确性。副本不需要子分区的弱证书，除非检查点 c 中的子分区的值已被丢弃。下面的段落更详细地描述了状态转移机制。

一个副本 i 多播⟨FETCH, l, x, lc, c, k, i⟩$_{\alpha i}$ 给所有其他副本，以获得树中 l 级别的索引 x 的分区的信息。这里 lc 是分区的最后一个检查点 i 知道的的序列号，c 是 nil 或者是它指定的 i 正在从副本 k 中寻找的序列号为 c 的分区的值。

当一个副本 i 决定发起状态转移时，它将使用等于其最后一个检查点号的 lc 来为根分区多播一个 FETCH 消息。当 i 知道检查点 c 处的分区的正确摘要时，c 的值不为零；例如，在视图变更完成后，i 会知道传播到新视图的检查点的摘要，但可能不会拥有它。i 还创建一个树的新（逻辑）拷贝来存储它获取的状态并初始化一个表 LC，其中它存储反映在新树中的每个分区的状态的最新检查点的数量。最初表中的每个条目都将包含 lc。

如果指定的副本 k 接收到⟨FETCH, l, x, lc, c, k, i⟩$_{\alpha i}$，并且它具有序列号 c 的一个检查点，它则发送回来⟨META-DATA, c, l, x, P, k⟩，其中 P 是每个子分区具有元组⟨x', lm, d⟩的一个集合（x'

是索引，d 是摘要，lm> lc）。由于 i 知道检查点 c 处的分区值的正确摘要，因此可以验证答复的正确性，而不需要证书甚至身份验证。这减轻了对其他副本的负担，当新视图中的请求处理的开始状态由单个正确的副本持有时，在视图变更中提供活力就非常重要。

除了指定回复者之外的副本，只有在它们有一个大于 lc 和 c 的稳定的检查点时才会回复 FETCH 消息。他们的回复类似于 k 的回复，除了 c 被它们的稳定检查点的序列号替换，并且消息包含一个 MAC。这些答复是必要的，以保证当副本已经丢弃了 i 要求的一个特定的检查点时的进展。

副本 i 重传 FETCH 消息（每次选择不同的 k），直到它收到来自某个 k 或一个存在对于相同序列号 c′ （大于 lc 和 c）的相同子分区值的同样新鲜的响应弱证书的有效答复。然后将其对（l，x）的每个子分区的摘要与获取的信息中的摘要进行比较；它会为存在差异的子分区多播一条 FETCH 消息，并将 LC 中的值设置为最新的子分区的 c（或 c'）。由于 i 在检查点 c（或 c'）处了解每个子分区的正确摘要，所以它可以使用优化协议来使用这些摘要来获取它们，以检查它们是否正确。

协议在树上递归，直到 i 为过期页面发送 FETCH 消息。页面像其它分区一样被获取，除了 META-DATA 的回复包含页面的摘要和最后修改序列号，而不是子分区，而且指定的回复者发送回⟨DATA, x, p⟩消息。这里 x 是页面索引，p 是页面值。该协议对其他副本几乎不产生开销；只有一个副本回复整个页面，它甚至不需要计算消息的 MAC，因为 i 可以使用它已经知道的摘要来验证回复。

当 i 获取一个页面的新值时，它会更新页面的状态、摘要、最后修改的序列号的值、以及 LC 中对应的页面值。然后协议升到其父级，并获取另一个缺失的兄弟节点。当获取到所有的兄弟节点后，它检查父分区是否一致。分区与序列号 c 一致，如果 c 是其子分区的 LC 中所有序列号的最小值，并且 c 大于或等于其子分区中最后修改序列号的最大值。如果父分区不一致，协议会发送该分区的另外一个获取。否则，协议再次上升到其父级，并获取缺失的兄弟节点。

当协议访问根分区并确定它对于一些序列号 c 是一致的时候，该协议终止。然后，副本可以开始处理序列号大于 c 的请求。

由于状态转移与其他副本上的请求执行同时发生而且其他副本可以自由进行检查点的垃圾回收，因此副本可能需要一些时间来完成协议；例如，每次它获取一个丢失的分区，它会收到一条关于稍后修改的信息。如果服务器操作更改数据的速度比传输速度更快，则一个过期的副本可能永远不会赶上它。所描述的状态转移机制可以足够快地传输数据，这对大多数服务器来说都不是什么问题。可以通过从不同副本并行获取页面来提高传输速率，但是目前这还尚未实现。此外，如果副本获取状态是实际需要的（因为其他的已经失败），系统将等待它追赶上来。

6.2.3 状态检查（State Checking）。在复原后，有必要确保副本的状态是正确的并且是最新的。这是通过使用状态转移机制来获取过期页面并获得更新的分区的摘要来完成的；恢复副本使用这些摘要来检查其分区的拷贝是否正确。

恢复副本首先计算所有元数据的分区摘要，假设页面的摘要与其存储的值相匹配。然后，除了将用于每个元数据分区的第一个 FETCH 消息中的 lc 的值设置为-1 之外，它启动如上所述的状态转移。这确保 META-DATA 回复包括所有子分区的摘要。

副本处理之前所述的 FETCH 消息的回复，但是不会忽略最新分区，它会检查分区摘要是

否与分区树中记录的相匹配。如果没有，分区会像过期了一样排队等待取出；否则，分区会排队等待检查。

分区检查与等待获取回复花费的时间重叠。副本通过计算每个分区的页面的摘要并将这些摘要与分区树中的摘要进行比较来检查一个分区。这些不符合摘要的网页排队等待取出。

7. THE BFT LIBRARY

该算法已被实现为具有简单接口的通用程序库。该库可用于提供不同服务器的拜占庭容错版本。第 7.1 节描述了该库的实现，第 7.2 节介绍了它的接口。我们使用里该库来实现拜占庭容错的 NFS 文件系统，这在第 7.3 节中有描述。

7.1. Implementation

该库使用无连接通信模型：使用 UDP [Postel 1980]实现节点之间的点对点通信，并且使用 IP 多播上的 UDP 实现副本组上的多播[Deering and Cheriton 1990]。每个服务器上都有一个包含所有副本的 IP 多播组。客户端不是此多播组的成员（除非它们也是副本）。

该库用 C ++实现。我们使用了一个带有和附录中 formalization of the algorithm 中与 I / O automaton 代码非常相似的结构的事件驱动的实现。副本和客户端是单线程的，其代码被构造为一组事件处理程序。该集合包含每个消息类型的处理程序和每个定时器的处理程序。每个处理程序对应于正式化中的一个输入操作，并且还有与内部操作对应的方法。代码和正式化之间的相似性是故意设计的，重要的是：它帮助确定了代码中几点错误和正式化中的遗漏。

事件处理循环工作如下。副本和客户端在选择呼叫中等待一条消息到达或等待定时器截止时间到达，然后它们调用相应的处理程序。处理程序执行类似于正式化中的相应操作的计算，然后调用与前提条件为真的内部操作相对应的任何方法。0 处理程序从不阻塞消息的等待。

我们使用了具有 1,024 位模数的 Rabin-Williams 公钥密码系统的 SFS [Mazieres et al. 1999]实现，建立了 128 位会话密钥。然后用使用这些密钥和 UMAC32 [Black et al. 1999]计算的消息认证代码来验证所有消息。消息摘要使用 MD5 [Rivest 1992]计算。

公共密钥加密签名和加密消息的实现，分别在 Bellare 和 Rogaway [1996]和[1995]中有所介绍。这些技术在随机 oracle 模型中是可靠的[Bellare and Rogaway 1995]。特别地，即使使用自适应选择的消息攻击，签名也是不可伪造的。UMAC32 在随机 oracle 模型中也是安可靠全的。MD5 仍然应该提供足够的安全性，并且可以以另一个散列函数（例如，SHA-1 [SHA1 1994]）轻易替换，同时以一些性能下降为代价。

消息格式被设计成仅通过固定大小的报头就可以计算 MAC。这具有使得验证者计算的成本与副本数量呈线性增长的优点，独立于有效载荷大小（例如，独立于请求中的操作参数大小和 PRE-PREPARE 中的批处理大小）。

7.2. Interface

我们使用非常简单的接口将该算法实现成一个库（见图 7）。该库的某些组件在客户端上运行，而其他组件在副本上运行。

```
Client:
int Byz_init_client(char *conf);
int Byz_invoke(Byz_req *req, Byz_rep *rep, bool ro);

Server:
int Byz_init_replica(char *conf, char *mem, int size, proc exec,  proc nondet);
void Byz_modify(char *mod, int size);

Server upcalls:
int execute(Byz_req *req, Byz_rep *rep, Byz_buffer *ndet, int cid, bool ro);

int nondet(Seqno seqno, Byz_req *req, Byz_buffer *ndet);
```

Fig. 7. The replication library API.

在客户端，该库提供了使用配置文件初始化客户端的程序，该配置文件包含副本的公钥和 IP 地址，以及一个被调用以使操作执行的程序，invoke。最后一个程序执行协议的客户端，并在足够的副本作出响应时返回结果。该库还提供了一个拆分接口（图中未显示），具有单独的发送和接收调用以调用请求。

在服务器端，我们提供一个初始化程序，参数如下：一个具有副本和客户端的公钥 IP 地址的配置文件、存储服务器状态的内存区域、一个执行请求的程序、以及一个计算非确定性选择的程序。当我们的系统需要执行一个操作时，它会调用一个 execute 程序。该程序的参数包括一个具有请求操作的缓冲区及其参数 req，以及一个以操作结果 rep 填充的缓冲区。execute 程序使用服务器状态执行服务器指定的操作。当服务执行操作时，每次即将修改服务器状态时，它就调用 modify 程序通知该库有待修改的位置。这个调用使我们能够有效地维护检查点和计算摘要，如第 6.2.2 节所述。

此外，execute 程序将请求操作的客户端的标识符和一个指示是否使用只读优化处理请求的布尔值作为参数。服务器代码使用此信息执行访问控制，并拒绝修改状态但是被故障客户端标记为只读的操作。当 primary 接收到请求时，它通过调用 nondet 程序来选择对所请求操作的任何非确定性输入（例如，时间戳）。BFT 库确保副本对这个非确定性输入达成一致，并将其作为参数传递给 execute 程序调用[Castro 2001]。

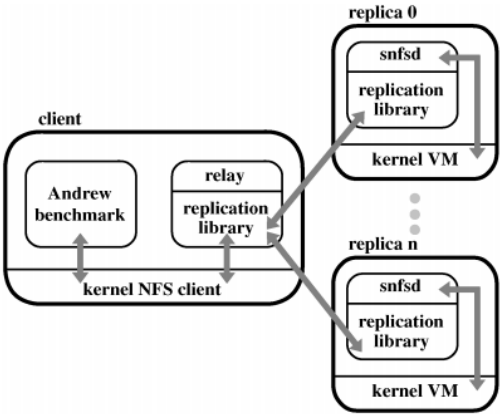

Fig. 8. BFS: replicated file system architecture.

7.3. BFS: A Byzantine-Fault-Tolerant File System

我们使用复制库实现了 BFS，一个拜占庭容错的 NFS [Sandberg et al. 1985]服务器。BFS 实现了 NFS 协议的版本 2。图 8 显示了 BFS 的架构。由容错 NFS 服务器导出的文件系统与任何常规 NFS 文件系统一样安装在客户端的机器上。应用程序进程未经修改地运行，并通过内核中的 NFS 客户端与挂载的文件系统进行交互。我们依靠用户级中继（relay）进程来调解标准 NFS 客户端和副本之间的通信。一个中继程序接收 NFS 协议请求，调用复制库的 invoke 程序，并将结果发送回 NFS 客户端。

每个副本使用复制库和我们的 NFS V2 守护程序运行用户级进程，我们将其称为 snfsd（即简单的 nfsd）。复制库接收来自 relay 的请求，通过调用 snfsd 进行交互，并将 NFS 回复打包到它发送给 relay 的复制协议回复。

我们使用固定大小的内存映射文件来实现 snfsd。所有文件系统数据结构（例如，inode，block 及其 free list）都在映射文件中。我们依靠操作系统管理内存映射文件页面的缓存，并将修改的页面异步写入磁盘。当前的实现使用 4 KB block 并且 inode 包含 NFS 状态信息加上 256 字节的用于存储目录中的目录条目的数据，文件中的区块指针，以及符号链接中的文本。目录和文件也可以以类似于 UNIX 的方式使用间接区块。

我们的实现确保所有状态机副本从相同的初始状态开始并且是确定性的，这是使用我们的协议实现的服务器的正确性的必要条件。primary 提出 time-last-modified 和 time-last-accessed 的值，并且副本选择提出的较大的值以及一个比早期请求选择的所有的值中最大一个还要大的值。primary 通过执行调用来计算非确定性选择来选择这些值，这样在这种情况下轻松返回 gettimeofday 的结果。

我们不需要同步写入来实现 NFS V2 协议语义，因为 BFS 通过像 Harp [Liskov et al. 1991]中实现的复制来达到修改的数据和元数据的稳定性。如果电源故障可能影响所有副本，则每个副本应具有不间断电源供应（UPS）。如在 Harp [Liskov et al. 1991]中完成的那样，在电源故障的情况下，UPS 将允许足够的时间让副本将其状态写入磁盘。

8. PERFORMANCE EVALUATION

BFT 库可用于实现拜占庭容错系统，但这些系统在实践中将不会被使用，除非它们的性能良好。本节介绍了评估这些系统性能的实验结果。

我们运行了几个基准来测量 BFS（拜占庭容错 NFS）的性能。结果表明，BFS 比每天大量用户都要使用并不被复制的 NFS 协议的生产实现速度在快 2％到慢 24%之间。此外，我们运行微基准测试，以独立于服务器的方式评估复制库的性能。我们介绍了一个详细的分析性能模型和实验来评估 Castro [2001]中每个优化的影响。

8.1. Mircobenchmarks

本节介绍微基准测试的结果。实验使用 8.1.1 节中的设置进行。8.1.2 和 8.1.3 节描述了使用四个副本测量简单复制服务的延迟和吞吐量的实验。在 8.1.4 节中，随着副本数量的增加，我们调查了其对性能的影响。这些部分的实验将在没有检查点管理、视图变更或恢复的情况

下评估性能。在 8.1.5 和 8.1.6 节中，我们分析了检查点管理和视图变更引入的性能开销。第 8.2.3 节研究了恢复性能。

8.1.1 实验设置（Experimental Setup）。实验运行在九台配有单个 Pentium III 处理器，512 MB 内存和 Quantum Atlas 10 K 18 WLS 磁盘的戴尔 Precision 410 工作站上。所有机器都运行无需 SMP 支持编译的 Linux 2.2.16-3 系统。七台机器的处理器时钟速度为 600 MHz，另外两台处理器的时钟速度为 700 MHz。除非另有说明，所有实验都运行在较慢的机器上。这些机器通过 100Mb / s 交换式以太网连接，并具有 3COM 3C905B 接口卡。该交换机是 Extreme Networks Summit48 V4.1。所有实验都运行在一个隔离网络上。

实验比较了一个简单服务器的两个实现的性能：一个实现，BFT，使用 BFT 库进行复制，另一个实现，NO-REP，不复制，并且在客户端和没有验证的服务器之间直接使用 UDP 进行通信。一个简单的服务器实际上是一个真实服务器的骨架：它没有状态，服务器操作从客户端接收参数并返回（零填充的）结果，但不执行计算。我们用 read-only（只读）和 read-write（读写）操作对不同参数和结果大小进行了实验。重点要注意，这是最坏情况的比较；在实际的服务器中，客户端和服务器上的计算或 I / O 将放缓 BFT 库带来的减速效果（如第 8.2 节所示）。

库配置如下：检查点之间的周期为 128 个序列号，日志的大小为 256 个序列号，请求批处理的窗口大小为 1。

8.1.2 延迟（Latency）。我们测量了当单个客户端访问服务器时调用一个操作的延迟。结果是通过在三次单独的运行中定时进行大量调用获得的。我们报告三次运行的平均值。标准差总是低于报告值的 3%。图 9 显示了在将参数大小固定为 8-B 时，随着操作结果的增加，调用复制服务器的延迟。其中一个是结果大小随时间变化的图，另一个是随着相对于 NO-REP 的 BFT 减速变化的图。

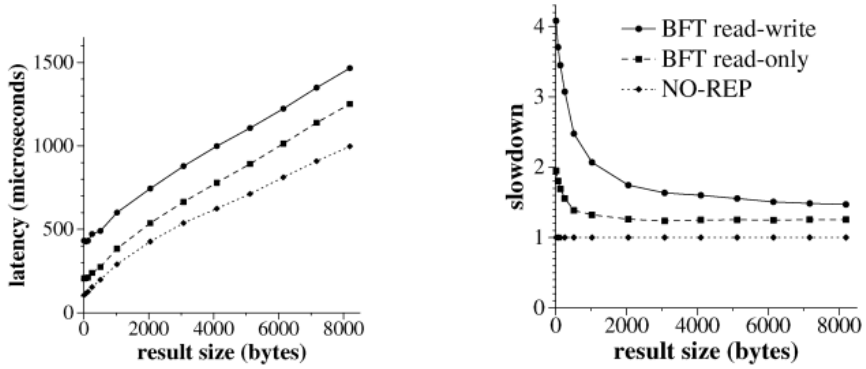

Fig. 9. Latency with varying result sizes: absolute times and slowdown relative to NO-REP.

图 10 显示了在保持结果大小固定为 8 个字节的情况下，随着操作参数的大小增加，调用复制服务器的延迟。这两个图都含有 read-write（读写）和 read-only（只读）操作的结果。

该库介绍了相对于 NO-REP 的显着的减速，但是随着操作参数或结果大小的增加，这种减速快速下降。例如，读写操作的减速从 8-B 结果的 4.08 减少到 8-KB 结果的 1.47，并且由只读优化的 1.95 下降到 1.25。开销的两个主要来源是（请求和回复的）摘要计算以及由复制协议引起的附加通信。MAC 计算的成本可以忽略不计。

由于发送回复（或请求）的通信时间和摘要回复（或请求）的时间随结果（或参数）大

小而增长，所以延迟增加。在我们的实验设置中，通信时间增加了 91 毫微秒/字节，摘要计算时间增加了 24 毫微秒/字节。由于 NO-REP 的延迟也增加了 91 毫微秒/字节，因此减速随着结果或参数大小的增长而降低，直到降到一条(91 + 24) / 91 = 1.26 的渐近线。

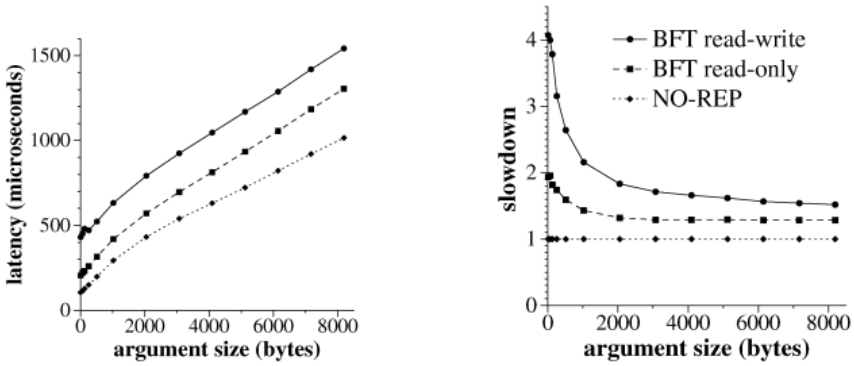

Fig. 10. Latency with varying argument sizes: absolute times and slowdown relative to NO-REP.

只读优化在降低 BFT 库带来的减速方面非常有效。它通过消除准备请求的时间来提高性能。随着参数或结果大小的增加，这个时间都不会改变。因此，只读优化提供的加速度随着参数或结果大小的增加而减小到零。例如，它使用 8-B 参数将延迟减少了 52%，而对于 8-KB 参数则将延迟只减少了 15%。

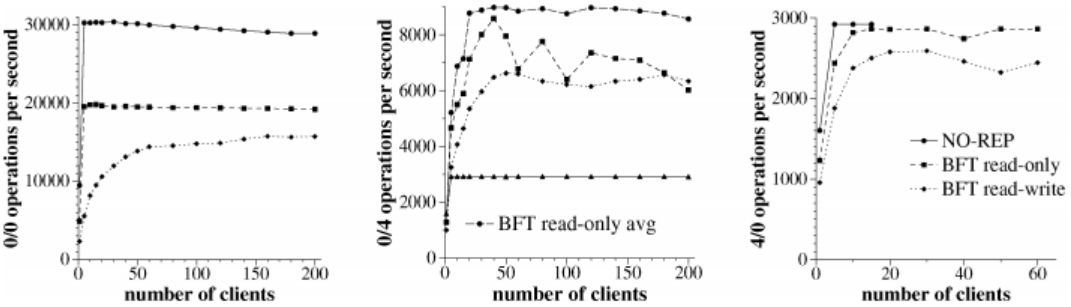

Fig. 11. Throughput for operations 0/0, 0/4, and 4/0.

8.1.3 吞吐量（Throughput）。本节报告了测试 BFT 和 NO-REP 作为访问简单服务的客户端数量的函数的吞吐量的实验结果。客户端进程均匀分布在五台客户机上[2]。我们测量了具有不同参数和结果大小的操作的吞吐量。每个操作类型由 a / b 表示，其中 a 和 b 是参数和结果的大小，以 KB 为单位。

实验运行如下：所有客户端进程几乎同时启动调用操作；每个客户端进程执行 3K 操作（其中 K 是一个大数字），并测量执行中间 K 操作的时间。吞吐量计算为 K 乘以客户端进程的数量，然后除以完成 K 操作的最大时间（取决于所有客户端）。这种方法提供了一个保守的吞吐量测量：它考虑到客户端被不公平对待并且需要更长时间来完成 K 次迭代的情况。报告的每个吞吐量值是至少三次独立运行的平均值。

图 11 显示了操作 0/0，0/4 和 4/0 的吞吐量结果。标准差低于报告值的 7%，除了只读操作 0/4（高达 18%）。

操作 0/0 的瓶颈是服务器的 CPU。由于增加了 CPU 负载的额外消息和加密操作，BFT 具有比 NO-REP 低的吞吐量。BFT 的读写操作的吞吐量低了 52%，只读操作低了 35%。只读优

[2] Two client machines had 700-MHz PIIIs but were otherwise identical to the other machines.

化通过消除准备批次请求的成本来提高吞吐量。读写操作的吞吐量随着客户端数量的增加而提高，因为准备批次的请求的成本在批量的大小上进行分摊。因为我们限制一个批次中的请求数量来防止 denial-of-service 攻击，使得吞吐量达到饱和。

对于 0/4 操作，BFT 具有比 NO-REP 更好的吞吐量。NO-REP 的瓶颈是链路带宽（12 MB/s）；它每秒执行大约 3000 次操作。BFT 通过摘要回复优化实现更好的吞吐量：客户端可以从不同的副本并行获得 4-KB 结果的回复。BFT 通过读写操作实现每秒最多 6625 次操作（26 MB/s）和只读优化实现每秒 8698 次操作（34 MB/s）的吞吐量。BFT 的瓶颈是副本的 CPU。

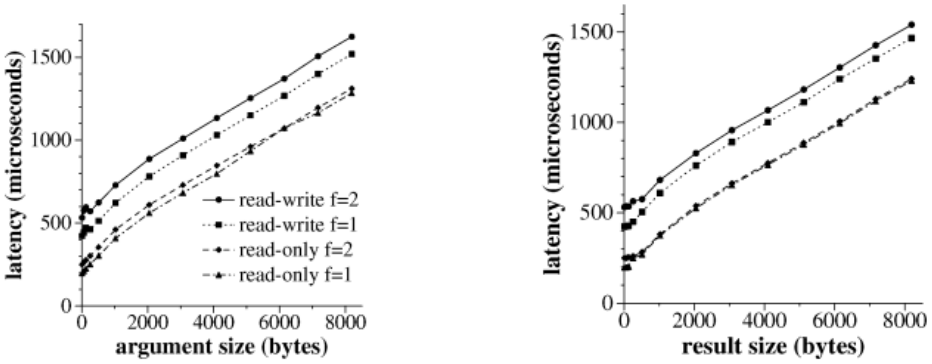

Fig. 12. Latency with varying argument and result sizes with $f = 2$.

只读优化的操作 0/4 的吞吐量非常不稳定，因为系统对所有客户端不是公平的；完成 K 次操作的最大时间差异很大。计算这些操作的平均时间保持稳定，如使用此时间计算的标记为“avg”的吞吐量值所示。

NO-REP 和 BFT 的操作 4/0 的瓶颈是通过网络获取请求的时间。由于链路带宽为 12 MB/s，可达到的最大吞吐量为每秒 3000 次操作。NO-REP 每秒最大吞吐量可达 2921 次，BFT 读写操作可达 2591（比 NO-REP 少 11％），只读优化为 2865（比 NO-REP 少 2％）。由于丢失请求消息，NO-REP 没有超过 15 个客户端的点；NOREP 直接使用 UDP，而且不重传请求。

8.1.4 配置更多副本（Configuration with More Replicas）。前面部分的实验运行在具有四个副本的配置中，这可以容忍一个故障。我们认为，这种可靠性水平对于大多数应用来说是足够的。但一些应用程序将具有更严格的可靠性要求，并且需要在具有更多副本的配置中运行。因此，重要的是了解当副本数量增加时是如何影响使用 BFT 库实现的服务器的性能的。图 12 比较了使用四个副本（f = 1）和七个副本（f = 2）调用复制服务器的延迟时间：第一个图显示了作为参数大小的函数的延迟，第二个图显示了作为结果函数大小的延迟。标准差始终低于报告值的 2％。在这两种配置中，所有的副本都拥有一个 600 MHz 的 Pentium III 处理器，客户端拥有一个 700 MHz Pentium III 处理器。

结果显示，副本数量增加到七个产生的减速是很低的。读写操作的最大减速为 30％，只读操作的最大减速为 26％。此外，随着参数或结果大小的增加，减速将快速下降。例如，参数大小为 8 KB，读写操作的减速仅为 7％，只读优化则为 2％。随着参数大小的增加，减速将降低，因为添加副本引入的开销与此大小无关。摘要回复优化使得添加副本产生的开销独立于结果大小，这就解释了为什么随着结果大小的增加，减速也会降低。

8.1.5 检查点管理（Checkpoint Management）。前几节的实验使用了一个没有状态的简单服务器。这些实验中唯一的检查点管理开销是由于将最后的回复存储在发送到每个客户端的读写操作中产生的。本节分析了使用添加状态的简单服务的修改版本的检查点管理引入的性

能开销。新服务器中的状态是由使用 256 MB 的内存映射文件的副本实现的一组持久的连续页面。服务器操作可以读取或写入这些页面。实验以一个客户端和四个副本来运行。本节介绍了测量创建检查点的时间和状态传输的时间的实验结果来将副本保持更新。

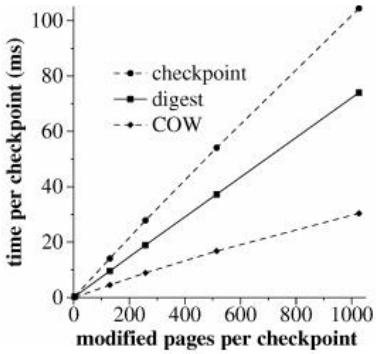

Fig. 13. Checkpoint cost with a varying number of modified pages per checkpoint epoch.

检查点创建（Checkpoint Creation）。检查点使用第 6.2 节中描述的技术创建。在我们的实验设置中，状态分区树有四个级别，每个内部节点有 256 个子节点，页面（即树的叶子）有 4 KB 大小。在两个检查点之间执行的请求被称为在相同的检查点时期。

检查点创建的成本有两个组成部分：执行 copy-on-write（COW）的时间和计算检查点摘要的时间。图 13 显示了我们在这些时间中测量的值，每个检查点时期具有不同数量的修改过的页面。结果表明，执行 copy-on-write 的时间和计算摘要的时间都随着检查点时期修改的不同页面的数量呈线性增长：摘要每页大概需要 72 μs，拷贝一个页面需要 29 μs。

检查点创建的成本可以表现为在变化率较高时运行一个操作的平均成本的很大一个部分。可以通过惰性计算检验点摘要来提高性能。该协议可以修改为不在 CHECKPOINT 消息中发送检查点摘要。因此，检查点摘要只需要在视图更改或状态转移之前进行计算。这可能在正常情况下，以降低视图变更和状态转移的速度为代价，大幅度减少开销。

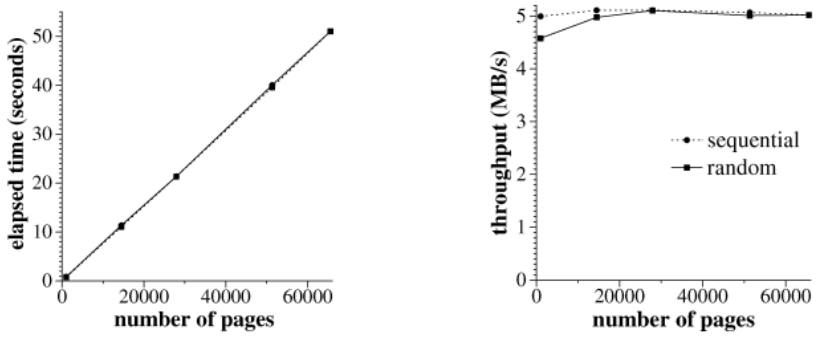

Fig. 14. State transfer latency and throughput.

状态转移（State Transfer）。我们还进行了实验来测量完成状态转移的时间。一个客户端调用了修改一定数量的页面（m）的操作。然后客户端被停止，其中一个备份从其初始状态重新启动。我们测量完成状态转移的时间，使该备份在空闲系统中保持最新。对随机选择的页面和依次选择的页面的几个 m 的值进行了实验。图 14 显示了完成状态转移及其吞吐量的经过时间。

结果表明，完成状态转移的时间与过时的页面数成正比。除了获取 1000 个随机页面时为 4.5 MB/s，吞吐量大约等于 5 MB/s。随机页面的吞吐量较低，因为需要获取更多的元数据

信息，但是这个额外的开销与获取大量页面的时间相差很大。完成状态转移的时间主要由获取数据页面的时间和计算其摘要以检查正确性的时间主导。

如果状态修改率大于状态转移吞吐量，则一个过期的副本可能无法追赶上来。此问题可能会降低可用性：如果出现故障，系统将停止处理客户端请求，直到过期副本可以完成状态传输。有几种方法可以来改善这个问题。可以通过从所有副本并行获取页面来提高状态传输的吞吐量；这应该提高链路带宽（12 MB/s）的吞吐量。此外，副本可以优先处理 FETCH 请求。

8.1.6 视图变更（View Changes）。迄今为止描述的实验分析了当系统没有故障时的性能。本节研究视图变更协议的性能。它测量从副本发送一条 VIEW-CHANGE 消息到准备好开始在新视图中处理请求的时间。这一次不仅包括接收和处理 NEW-VIEW 消息的时间，而且包括获取任何丢失的请求的时间，并且如果需要的话，还有选择为新视图中的请求处理的初始点的检查点。

Table I. Average View Change Time with Varying Write Percentage

	idle	10%	50%
View-change time (μs)	575	4162	7005

我们测量了使用 256 MB 状态，4 KB 页面和四个副本的简单服务器来完成视图变更协议的时间。有一个客户端调用了两种类型的操作：一个返回页面值的只读操作和将页面写入状态的写入操作。客户随机选择操作类型和页面。视图变更是由单独的进程触发的，该进程同时多播将导致所有副本在适当时间移动到下一视图的特殊消息。

表 I 显示了完成一个空闲系统的视图变更的时间，以及客户端以 10 和 50%概率执行写入操作的时间。对于每个实验，我们对每个副本的 128 个视图变更进行了计时，并显示了所有副本的平均值。

副本从不在空闲系统中预准备任何请求。因此，这种情况代表完成一次视图变更的最短时间。这个时间只比简单服务器上 0/0 操作的延迟时间高 34%。副本处理客户端请求时视图变更时间增加，因为 VIEW-CHANGE 消息包含有关副本在之前视图中发送的消息的信息。

视图变更时间从 10%增加到 50%的写入主要是由于一次视图变更花费了 607 毫秒才能完成，因为副本已经过期并且必须先获取缺少的检查点才能在新视图中开始处理新的请求；这种类型的事件的概率随状态修改的速度增加而增加。

由于我们的库中的视图变更协议的成本很小，我们可以将视图变更超时设置为一个很小的值（例如不到一秒）以提高可用性，而不会因不必要的视图变更而导致性能下降。

8.2. File System Benchmarks

接下来，我们介绍一组评估实际服务器，BFS，性能的实验结果。实验比较了 BFS 与 NFS 的另外两种实现的性能：NO-REP，它与 BFS 相同，不同之处在于它不是复制的；NFS-STD，即在服务器上 Ext2fs Linux 中的 NFS V2 实现。第一个比较使我们能够在实际服务器的实现中准确地评估 BFT 库的开销。第二个比较表明，BFS 是实用的：其性能类似于 NFS-STD 的性能，每天都有许多用户使用它。由于在 Linux 上 NFS 的实现不能保证在回复客户端之前的数据和

元数据的修改的稳定性（根据 NFS 协议[Sandberg et al. 1985]的要求），我们还将 BFS 与 NFS-DEC（其在 Digital UNIX 上的 NFS 实现并提供正确语义）进行了比较。

该部分从实验设置的描述开始。然后，它评估了无需视图变更和主动恢复的 BFS 的性能，并以主动恢复成本分析结束。

8.2.1 实验设置（Experimental Setup）。评估 BFS 的实验使用了第 8.1.1 节中描述的设置。它们运行了两个著名的文件系统基准：修改过的 Andrew 基准[Ousterhout 1990; Howard et al. 1988]和 PostMark [Katcher 1997]。

修改过的 Andrew 基准模拟了软件开发的工作量。它有几个阶段：（1）递归地创建子目录；（2）复制源树；（3）检查树中所有文件的状态而不检查其数据；（4）检查所有文件中的每个数据字节；（5）编译并链接文件。

不幸的是，Andrew 对于今天的系统来说太小了，以至于不能执行 NFS 服务。所以我们将基准的大小增加了 n 倍，如下所示：阶段 1 和阶段 2 创建源树的 n 个拷贝，其他阶段在所有这些副本中操作。我们运行了一个 n 等于 100 的 Andrew 的版本，Andrew100，和另一个 n 等于 500 的版本，Andrew500。BFS 在内存映射文件中构建一个文件系统。我们在 205 MB 的文件系统文件中运行 Andrew100，在 1 GB 的文件系统文件中运行 Andrew500；这两个基准都占据了这些文件的 90％以上。Andrew100 适合在客户端和副本的内存中，但是 Andrew500 不会。

PostMark [Katcher 1997]模拟互联网服务提供商的负载。它模拟电子邮件，网络新闻和基于 Web 的商务交易的组合所产生的工作量。基准测试通过在可配置的范围内创建大量随机大小的文件开始。然后在这些文件上运行大量的交易。每个交易由一对子交易组成：第一个子交易创建或删除文件，另一个读取文件或将数据附加到文件。每个子交易的操作类型是以均匀概率分布随机选择的。完成所有交易后，剩余的文件被删除。

我们配置了 PostMark，其中包含大小介于 512 字节和 16 KB 之间的 10,000 个文件的初始池。这些文件均匀分布在 130 个目录中。基准测试达到了 100,000 笔交易。

对于所有的基准测试和 NFS 实现，实际的基准测试代码在客户端工作站上运行，在 Linux 内核中使用标准的 NFS 客户端实现，具有相同的安装（mount）选项。这些基准测试选项中最相关的选项包括：UDP 传输，4,096 字节读取和写入缓冲区，允许回写的客户端缓存以及允许属性缓存。NOREP 和 BFS 都在客户端使用两个中继进程。

在 NFS V2 协议中的 18 个操作中，只有 getattr 是只读的，因为文件和目录的 time-last-accessed 的属性是由可能是只读的操作设置的，例如读取（read）和查找（lookup）。们修改了 BFS 和 NO-REP 不保留 time-last-accessed 属性，以便将只读优化应用于读取和查找操作。这种修改违反了严格的 UNIX 文件系统语义，但在实践中不太可能有不利影响。

8.2.2 无恢复的性能（Performance Without Recovery）。我们现在分析不带有视图变更和主动恢复的 BFS 的性能。我们首先介绍用四个副本运行的实验结果，然后介绍用七个副本获得的结果。

Andrew 基准（Andrew Benchmark）。图 15 显示了具有四个副本和一个客户端机器的配置的 Andrew100 和 Andrew500 的结果。我们报告三次基准测试的平均值。标准差总是低于报告平均值的 1％，除了第一阶段高达 33％。

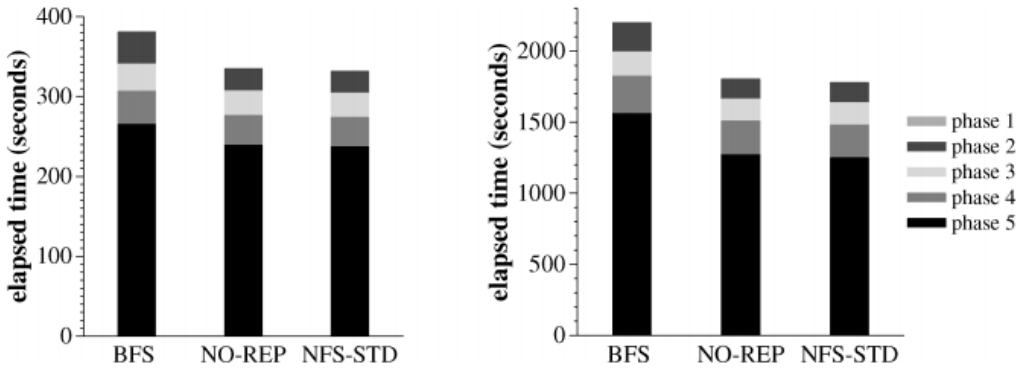

Fig. 15. Andrew100 and Andrew500: elapsed time in seconds.

BFS 和 NO-REP 之间的比较表明，对于该服务，拜占庭式容错的开销比较低，BFS 只需要运行 Andrew100 的时间增加 14%，运行 Andrew500 的时间增加 22%。这种减速比使用微基准测量的要慢，因为客户端花费了操作之间计算的经过时间的绝大部分，并且服务器平台上的操作执行了一些计算。另外，在 Andrew500 服务器上有大量的磁盘写入操作。基准阶段的开销不统一：前两阶段为 40%和 45%，后三阶段为 11%。造成这种情况的主要原因是客户端花费在操作之间计算的时间量不同。

与 NFS-STD 的比较表明，BFS 可以在实践中使用；完成 Andrew100 只需要多花费 15%，完成 Andrew500 只需要多花费 24%。如果 Linux 正确地实现了 NFS，性能差异会更小。例如，Castro [2001]的结果显示 BFS 比 Digital UNIX 中的 NFS 实现速度快了 2%，它实现了正确的语义。在 Linux 上实现 NFS 并不能确保修改的数据和元数据在响应客户端之前的稳定性（按照 NFS 协议的要求），而 BFS 通过复制确保稳定性。

邮戳（PostMark）。图 16 显示了使用 PostMark 测量的吞吐量。结果是三次运行的平均值，标准差低于报告值的 2%。在这个基准测试中拜占庭容错的开销更高：BFS 的吞吐量比 NO-REP 低 47%。这是通过减少客户端相对于安德鲁的计算时间来解释的。有趣的是，BFS 的吞吐量仅比 NFS-STD 低 13%。在这个工作负载中，由 NFS-STD 执行的磁盘访问次数的增加抵消了较高的开销。

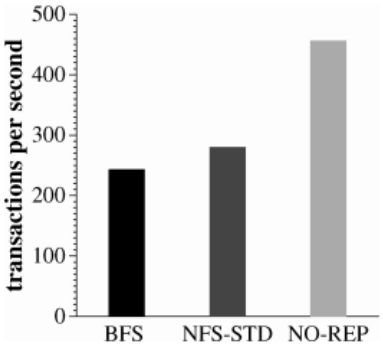

Fig. 16. PostMark: throughput in transactions per second.

更多副本（More Relpicas）。我们还运行了一个带有 7 个副本（f = 2）的配置的 Andrew100。所有副本都有一个 600 MHz 的 Pentium III 处理器，客户端有一个 700 MHz 的 Pentium III 处理器。结果显示，通过将副本数量从四个增加到七个，从而提高系统的弹性并不会显着降低性能：具有 f = 2 的 BFS 仅比具有 f = 1 的慢 3%。考虑到之前部分的微型基准结果，这个结果是可以预测的。

8.2.3 具有恢复的性能（Performance with Recovery）。频繁的主动恢复和密钥变更通过降低漏洞窗口来提高故障恢复能力，但也会降低性能。我们运行 Andrew 来确定可以实现的最小漏洞窗口，而不会导致重复数据的重叠。然后我们配置复制的文件系统来实现这个窗口，并测量相对于没有恢复的系统的性能下降。

除了我们在软件中模拟安全协处理器，只读存储器和看门狗定时器之外，主动恢复机制的实现是完整的。我们也在模拟快速重启。LinuxBIOS 项目[Minnich 2000]一直在尝试用 Linux 取代 BIOS。他们声称能够在 35 秒内重新启动 Linux（0.1 秒内核运行，34.9 内核在/etc/rc.d 中执行脚本）[Minnich 2000]。这意味着在适当配置的机器中，我们应该能够在不到一秒的时间内重启。副本通过睡眠 1 或 30 秒来模拟重启，并调用 msync 来使服务状态页面失效（这将在下次访问时强制从磁盘读取数据）。

恢复时间（Recovery Time）。完成恢复所需的时间决定了可以实现的不存在重叠的最小漏洞窗口。我们测量了 Andrew100 和 Andrew500 的回复时间，重启时间为 30s，密钥变更之间的时间为 T_k = 15s。

Table II. Andrew: Maximum Recovery Time (seconds)

	Andrew100	Andrew500
save state	2.84	6.3
reboot	30.05	30.05
restore state	0.09	0.30
estimation	0.21	0.15
send new-key	0.03	0.04
send request	0.03	0.03
fetch and check	9.34	106.81
total	42.59	143.68

表 II 列出了在两个基准中恢复副本的最长时间的细目。由于检查状态的正确性和通过网络获取丢失更新以使恢复副本保持最新的过程是并行执行的，所以表 II 为两者提供了一个单行。该行标记为“restore state”，只负责从磁盘读取日志；服务器状态页面在被检查时从磁盘按需读取。

恢复时间最重要的组成是将副本的日志和服务器状态保存到磁盘的时间，重新启动的时间以及检查和获取状态的时间。其他的都是微不足道的。重启的时间是 Andrew100 的主要组成，并且在 Andrew500 的大部分恢复时间内检查和获取状态帐户，因为该状态更大。

鉴于这些时间，我们设置在 Andrew100 中的看门狗监视器超时 T_w 为 3.5 分钟，在 Andrew500 中设置为 10 分钟。这些设置分别对应于 4 分钟和 10.5 分钟的最小漏洞窗口。我们还用 1 秒的时间重新启动 Andrew100 的实验，在这种情况下完成恢复的最长时间为 13.3 秒。这使 T_w 的一个 1.5 分钟的漏洞窗口设置为 1 分钟。

恢复必须快速以实现一个小的漏洞窗口。虽然目前的恢复时间很短，但是可以进一步减少这些时间。例如，可以通过定期将状态备份到通常写入保护的磁盘上，或者使用 copy-on-write 在可写磁盘上创建修改页面的拷贝来减少检查状态的时间。这样只有修改过的页面需要被检查。如果状态的只读副本经常（例如，每天）被更新到最新的状态，那么将可能扩展到非常大的状态，同时实现更低的恢复时间。

恢复开销（Recovery Overhead）。我们还评估了上一节中描述的实验设置中恢复对性能

的影响；图 17 显示了当漏洞窗口增加时，完成 Andrew100 和 Andrew500 所用的时间。BFS-PR 是主动恢复的 BFS。方括号内的数字是以分钟为单位的最小漏洞窗口。

结果表明，将频繁的主动恢复添加到 BFS 对性能的影响较小：在 Andrew100 中 BFS-PR [4] 比 BFS 慢了 16％，而 BFS-PR [1.5]只慢了 27％（即使每 15 秒有一个副本开始恢复）。Andrew500 主动恢复的开销更低：BFS-PR [10.5]比 BFS 慢 2％。

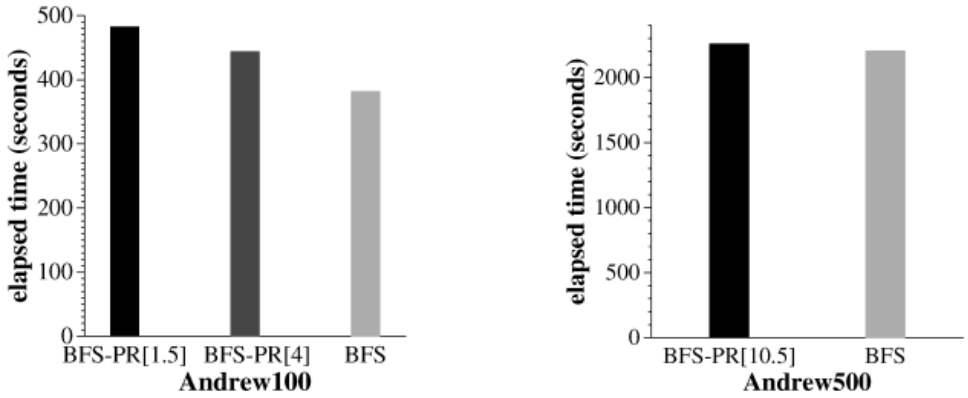

Fig. 17. Andrew: elapsed time in seconds with and without proactive recoveries.

恢复对性能的影响很小的原因有几个。最明显的是恢复是交错的，以致不会有超过一个的副本恢复；这允许剩余的副本继续处理客户端请求。但是，无论何时将恢复应用于当前的 primary，都必须执行视图变更，并且在视图变更完成之前客户端无法获得更多的服务。这些视图变更是廉价的，因为一个 primary 在其恢复开始之前多播一个 VIEW-CHANGE 消息，这会导致其他副本立即移动到下一个视图。

9. RELATED WORK

关于复制有大量的研究，但是早期的工作并没有提供足够的解决方案来构建可以容忍软件漏洞、操作失误或恶意攻击的系统。

9.1. Replication with Benign Faults

许多关于复制的研究都集中在容忍良性缺陷的技术（如 Alsberg and Day [1976], Gifford [1979], Schneider [1982], Oki and Liskov [1988], Lamport [1989], Liskov et al. [1991] and Keidar and Dolev [1996,1998]）：这项工作假设副本因停止或省略某些步骤而失败。这个假设对软件漏洞，操作失误或恶意攻击无效。例如，一个攻击者可以替换错误副本的代码以使其产生任意行为。此外，具有可变状态的服务器可能会在单个副本失败时返回不正确的回复，因为此副本可能会将损坏的信息传播给其他副本。因此，复制可能会降低对这些类型故障的恢复能力，因为不正确的服务行为的概率会随着副本的数量增加。

Viewstamped 复制 [Oki and Liskov 1988] 和 Paxos [Lamport 1989] 使用主备份（primary-backup）[Alsberg and Day 1976]和 quorum[Gifford 1979]技术的组合来容忍异步系统中的良性故障。他们使用一个 primary 分配序列号给请求，并且他们用一个视图变更协议来替换看起来有问题的 primary。两种算法都使用 quorum 来确保请求排序信息传播到新视图。BFT 从这两种算法中借鉴了这些想法，但是容忍拜占庭故障需要一个明显更复杂的协议。

9.2. Replication with Byzantine Faults

容忍拜占庭故障的技术[Pease et al. 1980; Lamport et al. 1982]对有缺陷组件的行为没有做出任何假设，因此它甚至可以容忍恶意攻击。然而，大部分早期工作（如 Pease et al. [1980], Lamport et al. [1982], Schneider [1990], Cristian et al. [1985], Reiter [1996], Garay and Moses [1998] and Khilstrom et al. [1998]）假设同步性。这个假设在一些系统中是合理的，例如航空电子控制[Wensley et al. 1978]。但是当恶意攻击者可以发起 denial-of-service 攻击来淹没处理器或虚假请求的网络时，这是特别危险的。

9.2.1 一致和共识（Agreement and Consensus）。一些一致性和共识算法在异步系统中容忍拜占庭故障（例如，Bracha and Taueg [1985], Canetti and Rabin [1992], Malkhi and Reiter [1996b], Doudou et al. [1999] and Cachin et al. [2000]）。但是，它们并没有为状态机复制提供完整的解决方案，而且它们中的大多数在实践中太慢而无法使用。

正常操作下的 BFT 协议与 Bracha and Toueg [1985]中的拜占庭一致算法类似。但是，这种算法不足以实现状态机复制：它保证非故障进程与 primary 发送的消息一致，但不能承受 primary 故障。

9.2.2 状态机复制（State Machine Replication）。我们的工作受到了 Rampart [Reiter 1994, 1995, 1996; Malkhi and Reiter 1996a]和 SecureRing [Kihlstrom et al. 1998] 的启发，它们也实现了状态机复制。但是，这些系统需要依靠安全性的同步假设。

Rampart 和 SecureRing 都使用具有动态组成员的组通信技术。他们必须从组中删除错误的副本以取得进展（例如，删除有故障的 primary 并选举一个新的），并执行垃圾回收。例如，一个副本需要知道，一条消息需要被组中所有副本接收到，这些副本才能丢弃这个消息，因此可能需要排除有故障的节点以丢弃消息。

这些系统依靠故障检测器来确定哪些副本有故障。然而，在异步系统中，故障检测器不可能是准确的[Lynch 1996]；也就是说，他们可能会把一个副本错误地分类为有故障的。由于正确性要求组中少于 1/3 的成员有故障，所以错误分类可以通过从组中删除非故障副本来损害正确性。这就打开了一个攻击途径：攻击者获得对单个副本的控制，但不会以任何可检测的方式改变其行为；那么它会减慢正确的副本或它们之间的通信，直到从组中排除足够的副本。甚至有可能这些系统没有任何妥协副本就会产生不正确的行为。如果将所有发送给客户端回复的副本从组中删除，并且剩余的副本从不处理客户端的请求，则会发生这种情况。

为了减少错误分类的概率，可以校准故障检测器以延迟将副本分类为故障副本。但是，对于可忽略的概率，延迟必须非常大，但这是不希望看到的。例如，如果 primary 实际上已经失败，那么该组将无法处理客户端请求，直到延迟过期，这会降低可用性。我们的算法不容易受到这个问题的影响，因为它只需要副本 quorum 之间的通信。由于总是有一个没有故障副本的 quorum 可用，BFT 从不需要从组中排除副本。

公钥密码系统是 Rampart 和 SecureRing 的主要性能瓶颈，尽管这些系统包括复杂的技术，以安全性或延迟为代价来降低公钥密码系统的成本。这些系统依靠公钥签名来正确地工作，并且不能使用对称密码来验证消息。BFT 使用 MAC 来认证所有消息，公钥密码仅用于交换对称密钥来计算 MAC。这种方法将性能提高了两个数量级，而不会损失安全性。

Rampart 和 SecureRing 提供了可用于实现恢复的组成员协议，但只在存在良性故障的情

况下。这些方法不能保证在拜占庭故障的情况下工作，原因有两个：如果一个没有故障的副本从需要被恢复的组中被删除，系统可能无法提供安全性；即使从组中删除之后，算法依赖于副本签名的消息，而且无法防止攻击者冒充他们控制的已删除副本。

我们在 Castro and Liskov [1999b]和 Doudou et al. [2000]中描述的算法与 BFT 类似。他们也在异步系统中正常工作，但他们依靠公钥加密来签署消息。因此表现不佳，而且不支持恢复。另外，Doudou et al. [2000]的算法不提供垃圾回收和状态转移机制。

9.2.3Quorum Replication。Phalanx [Malkhi and Reiter 1998a, b]及其后继者 Fleet[Malkhi and Reiter 2000]应用了 quorum 复制技术[Gifford 1979]，以实现异步系统中的拜占庭容错。这项工作不提供通用的状态机复制。相反，它提供了一个数据存储库，包含读取或写入各个变量的操作，并提供可以被客户端用来实现更复杂操作的共识对象。这使得 Fleet 更容易受到恶意客户端的攻击，因为它依赖于客户端对读写操作进行分组和排序，以便在服务器状态下保留任何不变量。正确的 Fleet 副本检查不变量是不平常的，因为在执行写入操作时，它们不一定与状态的值达成一致。

Fleet 提供了具有最佳弹性的算法（n > 3f 个副本以容忍 f 个故障），但是恶意客户端可以使用此算法使正确副本的状态产生偏差。为了防止这种情况，Fleet 需要 n > 4f 个副本。

Fleet 不提供故障副本的恢复机制。然而，它包含一个机制来估计系统中故障副本的数量[Alvisi et al. 1999]，以及基于这个估计值调整阈值 f 对系统容忍的故障数量的机制[Alvisi et al. 2000]。这很有趣，但不清楚它是否会在实践中发挥作用：聪明的攻击者可以使受损的副本看起来行为正确，直到它控制多于 f 个副本，然后以任何方式适应或响应都为时已晚。

Fleet 或 Phalanx 没有公布性能数据，但我们相信我们的系统速度更快，因为它在关键路径中的消息延迟更少，而且我们使用的是 MAC 而不是公钥密码。在 Fleet 中，写入要求执行三个消息往返操作，而 BFT 执行两次往返读写操作。更确切地说，Fleet 中的写入需要三个一对多消息交换和三个多对一消息交换，而在 BFT 读写操作中需要两个一对多交换，一个多对多交换，以及一个多对一的交换。Fleet 中大部分读取操作和 BFT 中的只读操作都需要一次往返操作，并涉及相同类型的消息交换。

另外，Fleet 中的所有通信都在客户端和副本之间进行。这减少了请求批量的机会，并可能导致延迟增加，因为我们预计在副本之间大多数配置中的通信将比与客户端的通信更快。

Fleet 中的方法提供了提高可扩展性的潜力：每个操作仅由副本的一个子集进行处理。然而，每个副本的负载随着 n（即$\Omega(1/\sqrt{n})$）缓慢下降。因此，我们认为客户端缓存和分区状态为多个副本组是更好的方法来实现大多数应用程序的可扩展性。

最近有一些关于扩充 Fleet 的工作，以支持状态机复制[Chockler et al. 2001]。这项工作使用类似于 BFT 的算法，客户端来担当 primary。该算法假定客户端是正确的，而且它假定对于活性延迟的最终时间界限，但在异步系统中是安全的。它需要具有公共密钥签名的 n > 5f 个的副本或没有签名的 n > 6f 个副本，并且每个操作需要四次往返。

COCA [Zhou et al. 2000] 采用 quorum 复制技术与主动恢复相结合的方式实现在线认证机构。与 BFT 一样，如果在任何漏洞窗口内的失败的副本数量少于 1 / 3，就会提供强大的安全性和活性保证。COCA 仔细地规定了证书操作的语义，以便能够提供活性而不依赖任何同步假设。由于 BFT 的一般性，BFT 必须依赖于一个弱同步的假设。

COCA 的主动恢复使用一个有趣的异步主动签名共享机制，以确保当副本失败并恢复时，

证书颁发机构的签名密钥不会受到影响。它不依赖于安全的协处理器来执行恢复，但可能需要管理员参与恢复受损副本。

COCA 提供 denial-of-service 攻击的防御措施，类似于 BFT [Castro 2001]。COCA 已经被实现，而且其性能已经在存在以及不存在 denial-of-service 攻击的情况下被评估。由于广泛使用公钥加密技术，性能比 BFT 差，但是有些加密技术无法用 COCA 中使用的证书认证规范来避免。

9.3. Other Related Work

以前有关拜占庭容错复制的工作尚未解决高效状态转移的问题。我们提出了一个有效的状态转移机制，使得以很低的性能下降频繁进行主动恢复。

SFS 只读文件系统[Fu et al. 2000]使用一种技术在副本和客户端之间传输数据，这与我们的状态转移技术相似。它们都基于 Merkle 树[Merkle 1987]，但只读 SFS 使用针对文件系统服务优化的数据结构。另一个不同之处在于，我们的状态转移在传输过程中处理对状态的修改，而它们的文件系统是只读的。我们在恢复过程中检查副本状态完整性的技术与 Blum et al. [1994]和 Maheshwari et al. [2000]的方法类似，除了我们从其他副本而不是从安全协处理器获得具有正确摘要的树。

Ostrovsky and Yung [1991]中介绍了一个能容忍多于 f 个故障的系统的概念，假设该系统在某个窗口内有不超过 f 个节点的故障。这个概念以前在同步系统中被用于秘密共享方案[Herzberg et al. 1995]，阈值加密[Herzberg et al. 1997]，以及最近的安全信息存储和检索[Garay et al. 2000]（提供单写单读复制变量）。但是我们的算法更具一般性；它允许异步系统中的一组节点实现一个任意的状态机。

10. CONCLUSION

我们社会对计算机日益增长的依赖性要求高度可用的系统提供正确的服务而不会中断。拜占庭故障，如软件漏洞，操作失误和恶意攻击是服务中断的主要原因。我们提出了一种新的复制算法和实现技术来构建可以容忍拜占庭故障并且可以在实践中使用的高度可用的系统。

本文介绍了 BFT，一种容忍拜占庭故障的状态机复制算法，前提是少于 1 / 3 的副本故障。BFT 提供线性化，这是一个强大的安全属性，不依赖于任何同步假设。另外，它保证活性，倘若最终消息延迟是有限的。BFT 提供安全性和活性，无论有多少拜占庭式的客户端。

本文还介绍了一种主动恢复机制，允许复制系统在系统的整个生命周期内容忍任何数量的故障，前提是少于 1 / 3 的副本在漏洞窗口内发生故障。可以频繁恢复副本，以将漏洞窗口缩短几分钟，同时对性能影响较小。该机制还提供了检测旨在增加窗口的 denial-of-service 攻击，并检测副本的状态何时被攻击者破坏。

BFT 已经作为一个通用的程序库被实现，并且具有简单的接口，本文描述了一个使用该库实现的服务：第一个 Byzantine 容错 NFS 文件系统，BFS。BFT 库和 BFS 表现良好。例如，具有四个副本的 BFS 与未复制的 NFS 协议的生产现相比，执行速度在提高 2％到降低 24％之间。这种良好的性能是由于几个优化产生的。最重要的优化是使用对称密码来验证消息。公

钥密码学是以前系统中的主要瓶颈，仅用于交换对称密钥。