Shared memory

Write operation for register r
Request event: < r, Write | v >
Indication event: < r, WriteReturn >

Read operation for register r
Request event: < r, Read >
Indication event: < r, ReadReturn | v >
v: return value

属性

共享内存寄存器是一种在多个进程之间同步信息的重要机制。它们的设计需要考虑到活性和安全性。

1. **活性（Liveness）**
   - **无等待（Wait-freedom）**：这是一种非常强的活性保证。它要求由任何正确的进程发起的每个操作最终都必须完成。无等待的特点在于，无论系统的其他部分如何延迟或故障，每个操作都保证会在有限的步骤内完成。
2. **安全性（Safety）**
   - **无并发写入时**：如果没有并发的写入操作发生，那么每个读操作应返回最后一次写入操作写入的值。这确保了在没有写入竞争的情况下，读取操作可以可靠地获得正确的数据。
   - **存在并发写入时**：当发生并发写入时，寄存器的行为可以根据不同的一致性语义来分类，包括：
     - **安全（Safe）*常规（Regular） 原子（Atomic）**

安全

1. **Safe（安全）语义**
   - **假设单一写入者**：安全语义通常假设只有一个进程可以写入寄存器，但可以有多个进程读取寄存器。
   - **非并发读的要求**：如果一个`read()`操作没有与任何`write()`操作并发发生，它必须返回最后一次写入的值。这确保了在没有并发写入的情况下，读操作可以可靠地获取最新的数据。
   - **并发读的情况**：如果一个`read()`操作与`write()`操作并发发生，那么它可以返回任何值。这意味着在并发情况下，读操作的结果可能是不确定的。它可能是最近的写入值，也可能是任何之前的值，甚至是一个无效的值。

总的来说，安全语义在没有并发写入的情况下提供了可靠的数据一致性，但在并发写入时不提供任何保证。

常规

在这种寄存器中，假设只有一个写入者（单写入者），但可以有多个读取者。

1. **非并发写的情况**
   - 如果`read()`操作没有与任何`write()`操作并发发生，它必须返回最后一次写入的值。这保证了在没有并发写入的情况下，读操作可以可靠地获取最新的数据。
2. **并发写的情况**

- 如果 `read()` 操作与 `write()` 操作并发发生，那么它可以返回两种值之一：
  - 最后一次写入的值：即使 `read()` 操作与 `write()` 操作并发，它仍可能返回最近一次 `write()` 操作写入的值。
  - 任何并发写入的值：`read()` 操作也可能返回正在进行的任何 `write()` 操作的输入参数值。

常规寄存器的这种特性使其比安全寄存器提供更强的一致性保证。在安全寄存器中，与写入操作并发的读取可能返回任何值，包括无效值。而在常规寄存器中，与写入操作并发的读取至少会返回一个有效的写入值（无论是最近的还是并发的）。

Regular register algorithms常规寄存器算法

setup

1. **设置**
   - 每个进程 `pi` 维护一个寄存器值的副本 `vi`。这个副本用于记录或反映寄存器的当前值。
2. **实现寄存器**
   - 实现寄存器的核心在于实现每个进程的 `Read()` 和 `Write()` 操作。
   - `Read()` **操作**：
     - 在返回一个 `Read()` 值之前，进程必须与其他进程进行通信。这是为了确保读取的值反映了最新的写入，或者是在并发写入情况下合适的值。
   - `Write()` **操作**：
     - 在执行 `Write()` 操作（即返回相应的确认ACK）之前，进程也必须与其他进程进行通信。这是为了确保写入的值能被系统中的其他进程正确识别和记录。
3. **假设**
   - 系统中存在可靠的点对点通信渠道。这意味着发送的消息总是能够可靠地到达目标进程，不会丢失或损坏。
   - 每个进程持有寄存器值的一个副本，这些副本共同构成了整个系统对寄存器状态的视图。

一种简单的实现共享内存寄存器的方法。基本思想是将一个进程（例如p1）作为集中式数据存储库来保存寄存器的值。这个进程类似于Dropbox或Github这样的集中式存储服务

```
At every process pi:

read()
send [R] to p1
wait to receive [v] from p1
return v

write(v) at writer
send [W,v] to p1
wait to receive [ACK] from p1
return ACK

At p1:
Init: v1 := ⊥
```

```
upon receive [R] from pi
    send [v1] to pi

upon receive [W,v] from pi
    v1 := v
    send [ACK] to pi
```

p1 might crash? then the register is not live (wait-free)

Only if p1 is always correctthen the register is regular and wait-free

Simple algorithm with failure detectors

属性

(1, N)-RegularRegister这种寄存器允许一个写入者（1）和多个读取者（N）。

1. **请求（Request）**
   - `< onrr, Read >`：这个请求调用寄存器的读操作。当一个进程需要读取寄存器中的值时，它发起这个请求。
   - `< onrr, Write | v >`：这个请求调用寄存器的写操作，将值`v`写入寄存器。这个操作仅由写入者进程执行。
2. **指示（Indication）**
   - `< onrr, ReadReturn | v >`：这个指示完成了读操作，并返回了寄存器中的值`v`。它标志着读操作的结束，并提供了读取的结果。
   - `< onrr, WriteReturn >`：这个指示完成了写操作。它标志着写操作的结束，但不提供任何返回值。
3. **属性（Properties）**
   - **ONRR1. 终止（Termination）**：如果一个正确的进程发起了一个操作，那么该操作最终会完成。这保证了无论系统状态如何，每个操作都能得到处理。
   - **ONRR2. 有效性（Validity）**：
     - 如果读操作与写操作不并发，它返回最后一次写入的值。
     - 如果读操作与写操作并发发生，它返回最后一次写入的值或当前正在写入的值。这提供了对并发写入的一定程度的容忍。

实现

```
Implements: (1, N)-RegularRegister, instance onrr.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl;
PerfectFailureDetector, instance P.

# 当(1, N)-常规寄存器初始化时触发的事件
upon event < onrr, Init > do
    val := ⊥;  # 初始化val为⊥（表示未定义）
    correct := Π;  # 初始化correct集合，包含所有进程
    writeset := ∅;  # 初始化writeset集合，用于跟踪确认写入的进程
```

```
# 当检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程

# 当有读取请求时触发的事件
upon event < onrr, Read > do
    trigger < onrr, ReadReturn | val >;  # 返回当前的val值

# 当有写入请求时触发的事件
upon event < onrr, Write | v > do
    trigger < beb, Broadcast | [WRITE, v] >;  # 广播写入请求[WRITE, v]

# 当通过尽力而为广播接收到写入请求时触发的事件
upon event < beb, Deliver | q, [WRITE, v] > do
    val := v;  # 更新val为接收到的v值
    trigger < pl, Send | q, ACK >;  # 向发送者q发送确认ACK

# 当通过完美点对点链接接收到确认ACK时触发的事件
upon event < pl, Deliver | p, ACK > do
    writeset := writeset ∪ {p};  # 将发送确认的进程添加到writeset集合

# 当所有正确的进程都确认了写入时触发的事件
upon correct ⊆ writeset do
    writeset := ∅;  # 清空writeset集合
    trigger < onrr, WriteReturn >;  # 完成写入操作
```

这个算法的主要思想是：

- 使用广播来传播写入请求，保证所有进程能够看到新的写入值。
- 当一个进程接收到写入请求时，它更新自己的本地副本，并向发送者确认。
- 写入者需要从所有正确的进程收到确认，才认为写入操作完成。

正确性

read()
Local

write()
Strong completeness property of P + reliable channels　【P+可靠信道的强完备性】

如果我们没有完美的故障检测器（FD）或同步机制，问题就变成了：我们是否仍然能够实现一个算法来容忍任意数量的（崩溃）故障?

A tight asynchronous lower bound（一个紧密的异步下界）

在异步系统中，进程之间没有固定的时间界限来完成操作或者进行通信。这意味着进程不能依赖于关于时间的假设来区分一个慢速的进程和一个已经崩溃的进程。以下是对该情况的进一步讨论：

1. **无等待（Wait-free）的异步实现的限制**

- 在一个无等待的异步实现中，每个正确的进程发起的操作都应该在有限步内完成，而不考虑其他进程的速度或者是否有进程崩溃。
- 为了实现无等待属性，任何一个进程都不能依赖于其他特定进程的响应。

2. **关于常规寄存器的下界定理**
   - 定理：在异步系统中，任何能够实现无等待属性的常规寄存器，都需要大多数进程是正确的。这里的“大多数”意味着超过一半的进程。
   - 假设：至少存在一个写入者和一个与写入者不同的读取者。
   - 这个定理的核心思想是，在没有全局时间界限的情况下，为了保证常规寄存器的一致性和可靠性，需要确保任何时刻至少有一半以上的进程是可靠的。这是因为在异步系统中，无法可靠地检测和区分崩溃的进程和正常的进程。

总结来说，在没有完美故障检测器的异步系统中实现无等待的常规寄存器需要大多数进程是正确的。这种要求是由于在异步系统中无法区分慢速进程和崩溃进程，因此需要确保任何时候都有足够多的进程能够响应来保持系统的一致性和可用性。

在`n=2f`的条件下，由于正确进程和可能故障的进程数量相等，无法确保在所有情况下都维持常规寄存器的一致性和有效性。这意味着在这种条件下实现常规寄存器是不可能的。

Algorithm without failure detectors

思路

没有故障检测器的情况下实现共享内存寄存器，核心思想是通过使用逻辑时钟（时间戳）和多数投票原则来确保一致性和可靠性。

1. **假设**
   - **写入者**：`p1`是唯一的写入者，而任何进程都可以是读取者。
   - **多数正确**：系统中大多数进程是正确的（其他的可以崩溃）。
   - **可靠的通信通道**：系统中的通信通道是可靠的，即消息总是能够被正确地传递。
2. **算法思路**
   - **逻辑时钟**：写入者`p1`维护一个逻辑时钟`wts`（写入时间戳），用于标记每次写入操作。
   - **本地副本和时间戳**：每个进程`pi`维护寄存器值的本地副本`val`和相应的时间戳`ts`，以及一个读取时间戳`rid`。
   - **向多数写入**：写入操作需要向大多数进程写入，以确保至少有一部分正确的进程接收到了新的值。
   - **从多数读取**：读取操作从大多数进程读取数据，以确保获得的是最新且一致的值。
   - **写入过程**：
     - 写入者`p1`增加`wts`，然后将带有`wts`的新值发送给所有进程。
     - 只有当`p1`从大多数进程接收到确认后，写入操作才算完成。
   - **读取过程**：
     - 读取者`pi`向所有进程发送读取请求，并等待响应。
     - `pi`从大多数进程收集响应，选择具有最新时间戳的值作为读取结果。

3. **算法的关键**
    - 这种方法利用了逻辑时钟来区分不同的写入操作，并通过与大多数进程的交互来确保一致性。即使有进程崩溃，由于保持了大多数进程的正确性，算法仍然能够保证寄存器操作的可靠性和一致性。

这种算法在分布式系统中特别有用，因为它不依赖于故障检测器，而是通过逻辑时钟和多数原则来协调进程之间的交互，从而实现了高度的容错性和可靠性。

实现

```
Implements: (1, N)-RegularRegister, instance onrr.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl.

# 当(1, N)-常规寄存器初始化时触发的事件
upon event < onrr, Init > do
    (ts, val) := (0, ⊥);  # 初始化时间戳ts和寄存器值val
    wts := 0;  # 初始化写时间戳wts
    rid := 0;  # 初始化读ID rid
    acks := 0;  # 初始化确认计数acks
    readlist := [⊥]N;  # 初始化readlist数组，用于收集读取响应

# 当有写入请求时触发的事件
upon event < onrr, Write | v > do
    wts := wts + 1;  # 增加写时间戳
    acks := 0;  # 重置确认计数
    trigger < beb, Broadcast | [WRITE, wts, v] >;  # 广播写请求[WRITE, wts, v]

# 当通过尽力而为广播接收到写请求时触发的事件
upon event < beb, Deliver | p, [WRITE, ts', v'] > do
    if ts' > ts then
        (ts, val) := (ts', v');  # 如果接收到的时间戳更大，则更新寄存器值
    trigger < pl, Send | p, [ACK, ts'] >;  # 向发送者发送确认[ACK, ts']

# 当通过点对点链接接收到确认时触发的事件
upon event < pl, Deliver | q, [ACK, ts'] such that ts' = wts do
    acks := acks + 1;  # 增加确认计数
    if acks > N/2 then
        trigger < onrr, WriteReturn >;  # 如果确认数超过一半，则完成写操作

# 当有读取请求时触发的事件
upon event < onrr, Read > do
    rid := rid + 1;  # 增加读ID
    readlist := [⊥]N;  # 重置readlist
    trigger < beb, Broadcast | [READ, rid] >;  # 广播读请求[READ, rid]

# 当通过尽力而为广播接收到读请求时触发的事件
upon event < beb, Deliver | p, [READ, r] > do
    trigger < pl, Send | p, [VALUE, r, ts, val] >;  # 向请求者发送当前值[VALUE, r, ts,
val]

# 当通过点对点链接接收到值响应时触发的事件
```

```
upon event < pl, Deliver | q, [VALUE, r, ts', v'] > such that r = rid do
    readlist[q] := (ts', v');  # 更新readlist
    if #(readlist) > N/2 then
        v := highestval(readlist);  # 从readlist中选择最高的时间戳对应的值
        readlist := [⊥]N;  # 重置readlist
        trigger < onrr, ReadReturn | v >;  # 完成读操作并返回值
```

正确性和表现

1. **活性（Liveness，特指无等待特性）**
   - **可靠的通信通道**：系统假设通信通道是可靠的，即发送的消息总是能够到达目标进程。
   - **大多数进程正确的假设**：系统中大多数进程是正确的。这意味着即使有部分进程发生故障，其他正确的进程仍然可以继续进行操作。
2. **常规性（Regularity）**
   - **无并发读写的情况**：
     - **非空多数交集**（Quorum）：系统中总有一部分进程（多数）能够反映最新的写入操作的结果。
     - **最后一次完成的写入**：最后一次完成的写入操作写入了多数进程。
     - **读操作从多数进程读取**：读操作从多数进程中获取数据。
     - **存在交集中的一个进程**：这个进程持有最新写入的值，该值具有最高的时间戳。
   - **存在并发读写的情况**：
     - **不会返回早于最后一次完全写入的值**（LCWO）：读操作不会返回比最后一次完全写入更旧的值。
     - **可能返回并发写入的值**：如果有与读操作并发的写入操作，读操作可能返回这些并发写入中的一个，这是可以接受的。
3. **性能**
   - **一个通信往返**：算法只需要一个通信往返（两个步骤）就能完成一个操作。
   - **O(N)消息复杂度**：每次操作的消息复杂度为O(N)，其中N是系统中的进程数。

Atomic registers(原子寄存器)

原子寄存器提供了一种强一致性模型，即使在存在读写并发和故障的情况下也是如此。

1. **原子性（Atomicity，或线性化性 Linearizability）**
   - **强一致性**：即使在读写操作并发执行以及存在故障的情况下，也能保证强一致性。
   - **等效于顺序执行**：原子（或线性化）执行等同于一个顺序执行且没有故障的执行。
   - **操作的完整性**：每个完成的操作看似在其请求和指示事件之间的某个瞬间执行。
   - **操作的失败处理**：每个失败的（写入）操作看起来要么是完整的，要么就像根本没有被调用过一样。
2. **原子寄存器的关键**：
   - **线性化点**：在每个原子执行中放置线性化点。线性化点是指在操作执行过程中的一个具体瞬间，使得每个操作好像在那一刻发生。

- **一致性视图**：所有进程观察到的操作顺序应该是一致的，即使操作在不同的进程间是并发执行的。
- **读写操作的处理**：对于写操作，要么完全执行，要么就好像没有执行过；对于读操作，它们应该返回最近一个线性化点的写入值。

在对比常规寄存器和原子寄存器时，区别在于它们处理连续读操作的方式，尤其是在涉及单个写入者且没有写入失败的情况下。

1. **常规寄存器与原子寄存器的区别**
   - 在原子寄存器中，所有操作都必须符合线性化顺序，就好像每个操作都在某个特定的瞬间完成一样。这意味着连续的读操作必须反映出写操作的这种序列化效果。
   - 在常规寄存器中，尽管每个非并发的读操作都返回最近的写入值，但它们不需要符合严格的线性化顺序。因此，在特定情况下，连续的读操作可能会返回不一致的结果。
2. **新旧反转（New-Old Inversion）**
   - **问题描述**：如果有两个连续的读操作（可能由不同的读取者执行），它们可能与同一个写操作重叠。在常规寄存器中，这可能导致所谓的"新旧反转"现象。
   - **具体表现**：在这种情况下，第一个读操作可能获得新的值（即刚写入的值），而紧接着的第二个读操作可能获得旧的值（即写入操作之前的值）。
   - **为什么发生**：这种情况发生的原因是常规寄存器不保证操作的线性化顺序，因此不同的进程可能在不同的时间点看到不同的系统状态。当读操作与写操作并发时，常规寄存器可能返回任何一个合法的写入值（最新的或任何一个并发的写入值）。
3. **对原子寄存器的要求**
   - 要使常规寄存器表现得像原子寄存器，必须确保两个连续的读操作不会重叠同一个写操作，或者至少它们必须以一种方式执行，使得不会发生新旧反转现象。
   - 在原子寄存器中，这种新旧反转是不允许的，因为所有操作都必须遵循一致的线性化顺序。

属性 (1, N)-AtomicRegister"

1. - `< onar, Read >`：这个请求启动对寄存器的读操作。当一个进程需要读取寄存器中的值时，会发起这个请求。
   - `< onar, Write | v >`：这个请求启动对寄存器的写操作，将值`v`写入寄存器。这个操作通常由指定的写入者进程执行。
2. **指示（Indication）**
   - `< onar, ReadReturn | v >`：这个指示标志着读操作的完成，并返回读取到的值`v`。
   - `< onar, WriteReturn >`：这个指示标志着写操作的完成。它不提供任何返回值，仅表明写入操作已经结束。
3. **属性（Properties）**
   - **ONAR1–ONAR2**：与"(1, N)常规寄存器"（简称ONRR）的属性ONRR1–ONRR2相同，包括：
     - **终止（Termination）**：如果一个正确的进程发起了一个操作，那么该操作最终会完成。
     - **有效性（Validity）**：一个不与写操作并发的读操作返回最后一次写入的值；一个与写操作并发的读操作返回最后一次写入的值或并发写入的值。
   - **ONAR3. 顺序性（Ordering）**：

- 如果一个读操作返回值`v`，紧接着的另一个读操作返回值`w`，则写入`w`的操作不会早于写入`v`的操作。这保证了读操作的顺序反映了写操作的实际顺序，即使这些操作在时间上是重叠的。

实现

要将一个常规的读取一次写入所有（read-one write-all）的寄存器算法改造为原子（atomic）算法，并保持原有的假设（任何进程都可能因崩溃而失败、完美的故障检测器P、可靠的通道），我们可以采用以下策略：

1. **常规算法的问题**：
   - 在常规的算法中，写入操作在所有进程中全局进行（write globally），而读取操作通常只在本地进行（read locally）。这可能导致不同进程间的数据状态不一致，尤其是在发生并发读写操作时。
2. **改进的核心思想**：
   - **全局读取**（Read globally）：改进的算法需要确保读取操作在全局范围内进行，意味着每次读取操作都要从多个进程中获取数据，以确定最新的写入值。
   - **如果读取成功，每个正确的进程都应该知晓**：这意味着读取操作不仅需要从多数进程中获取数据，而且还需要确保这些数据被广泛地传播给其他所有正确的进程。
3. **实现全局读取的方法**：
   - **强制读取值（或回写）全局化**（Impose the read value globally）：当一个进程完成读取操作并确定了最新的值后，这个值应该被回写（或强制应用）到所有进程中，确保系统中每个进程都有最新的数据状态。
   - 这可以通过在读取操作后进行额外的广播或通信步骤来实现，其中读取得到的值被发送给所有其他进程。
4. **实现步骤**：
   - 当进程执行读取操作时，它首先从一定数量的进程（通常是多数）中收集数据。
   - 然后，基于收集到的数据，确定当前的最新写入值。
   - 最后，将这个确定的值广播回所有进程，确保每个进程都更新到这个最新值。

```
Implements: (1,N)-AtomicRegister, instance onar.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl;
PerfectFailureDetector, instance P.

# 当原子寄存器初始化时触发的事件
upon event < onar, Init > do
    (ts, val) := (0, ⊥);  # 初始化时间戳ts和寄存器值val
    correct := Π;  # 初始化correct集合，包含所有进程
    writeset := ∅;  # 初始化writeset集合，用于跟踪确认写入的进程
    readval := ⊥;  # 初始化读取值readval
    reading := FALSE;  # 初始化读取状态标志reading

# 当检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程
```

```
# 当有读取请求时触发的事件
upon event < onar, Read > do
    reading := TRUE;  # 标记正在进行读取操作
    readval := val;  # 设置当前读取值为val
    trigger < beb, Broadcast | [WRITE, ts, val] >;  # 广播当前值以确认最新状态

# 当有写入请求时触发的事件
upon event < onar, Write | v > do
    trigger < beb, Broadcast | [WRITE, ts + 1, v] >;  # 广播新的写入请求

# 当通过尽力而为广播接收到写请求时触发的事件
upon event < beb, Deliver | p, [WRITE, ts', v'] > do
    if ts' > ts then
        (ts, val) := (ts', v');  # 如果接收到的时间戳更大，则更新寄存器值
    trigger < pl, Send | p, [ACK] >;  # 向发送者p发送确认

# 当通过点对点链接接收到确认时触发的事件
upon event < pl, Deliver | p, [ACK] > do
    writeset := writeset ∪ {p};  # 更新writeset集合

# 当所有正确的进程都确认了写入时触发的事件
upon correct ⊆ writeset do
    writeset := ∅;  # 重置writeset集合
    if reading = TRUE then
        reading := FALSE;  # 重置读取状态
        trigger < onar, ReadReturn | readval >;  # 完成读取操作并返回读取值
    else
        trigger < onar, WriteReturn >;  # 完成写入操作
```

这个算法的主要思想是：

- 每次写入操作都会通过广播来更新所有进程中的值，确保它们是最新的。
- 读取操作不仅仅是返回本地的值，而是先通过广播确认当前的值，然后返回这个经过确认的值。
- 通过在所有正确的进程中广播并确认写入操作，算法确保即使在并发读写和进程崩溃的情况下，读取和写入操作的结果都能保持一致性和原子性。

正确性

要证明“(1, N)-AtomicRegister”实例（简称onar）实现的原子性，关键在于证明不存在新旧反转（new-old inversion）。

1. **假设存在新旧反转来寻求矛盾**
    - 假设存在两个完整的连续读操作`rd1`和`rd2`，其中`rd1`在`rd2`之前完成，但是`rd1`返回了一个比`rd2`更新的值。
2. **分析读操作**
    - 假设`rd1`由进程`pi`发起，`rd2`由进程`pj`发起。
    - 一旦`rd1`完成，`pj`处的值`vj`的时间戳至少为`T1`，因为它至少已经知道了由`rd1`读取的值。
    - 因此，`pj`后来发起的`rd2`不可能返回一个时间戳`T2`小于`T1`的值，这就与我们的假设矛盾。

3. **性能**
    - **通信步骤**：该算法需要两个通信步骤来完成一次读或写操作。
    - **消息复杂度**：每次操作的消息复杂度为O(N)，其中N是系统中的进程数。

原子寄存器但又不依赖于故障检测器（FDs）

1. **基本假设**
    - **正确的多数**：系统中必须始终有大多数进程是正确的，即使存在崩溃故障。
    - **异步系统**：进程间没有固定的时间界限来完成操作或者进行通信。
    - **无故障检测器**：系统中不使用故障检测器。
2. **改进思路**
    - **扩展常规的单写多读（SWMR）算法**：
        - 常规的“多数算法”依赖于向大多数进程写入数据并从大多数进程读取数据来确保一致性。
    - **在读取者端加入回写（Write-back）阶段**：
        - 读取操作不仅仅是简单地返回读取到的值，而是在读取之后进行一个额外的回写阶段，将读取到的最新值回写到大多数进程中。这样可以确保即使其他进程没有参与最初的写入操作，它们也能获取到最新的数据。
    - **在所有地方等待多数响应**：
        - 无论是读操作还是写操作，都应确保收到来自大多数进程的响应。这样可以在没有故障检测器的情况下，确保每个操作都基于大多数进程的最新状态。

```
Implements: (1,N)-AtomicRegister, instance onar.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl.

# 当原子寄存器初始化时触发的事件
upon event < onar, Init > do
    (ts, val) := (0, ⊥);  # 初始化时间戳ts和寄存器值val
    wts := 0;  # 初始化写时间戳wts
    rid := 0;  # 初始化读ID rid
    acks := 0;  # 初始化确认计数acks
    readlist := [⊥]N;  # 初始化readlist数组，用于收集读取响应
    readval := ⊥;  # 初始化读取值readval
    reading := FALSE;  # 初始化读取状态标志reading

# 当有读取请求时触发的事件
upon event < onar, Read > do
    rid := rid + 1;  # 增加读ID
    acks := 0;  # 重置确认计数
    readlist := [⊥]N;  # 重置readlist
    reading := TRUE;  # 标记正在进行读取操作
    trigger < beb, Broadcast | [READ, rid] >;  # 广播读取请求[READ, rid]

# 当通过尽力而为广播接收到读取请求时触发的事件
upon event < beb, Deliver | p, [READ, r] > do
    trigger < pl, Send | p, [VALUE, r, ts, val] >;  # 向请求者发送当前值[VALUE, r,
ts, val]
```

```
# 当通过点对点链接接收到值响应时触发的事件
upon event < pl, Deliver | q, [VALUE, r, ts', v'] > such that r = rid do
    readlist[q] := (ts', v');  # 更新readlist
    if #(readlist) > N/2 then
        (maxts, readval) := highest(readlist);  # 从readlist中选择最高时间戳的值
        readlist := [⊥]N;  # 重置readlist
        trigger < beb, Broadcast | [WRITE, rid, maxts, readval] >;  # 广播回写请求
[WRITE, rid, maxts, readval]

# 当有写入请求时触发的事件
upon event < onar, Write | v > do
    rid := rid + 1;  # 增加读ID
    wts := wts + 1;  # 增加写时间戳
    acks := 0;  # 重置确认计数
    trigger < beb, Broadcast | [WRITE, rid, wts, v] > ;  # 广播写请求[WRITE, rid,
wts, v]

# 当通过尽力而为广播接收到写请求时触发的事件
upon event < beb, Deliver | p, [WRITE, r, ts', v'] > do
    if ts' > ts then
        (ts, val) := (ts', v');  # 如果接收到的时间戳更大，则更新寄存器值
    trigger < pl, Send | p, [ACK, r] >;  # 向发送者发送确认[ACK, r]

# 当通过点对点链接接收到确认时触发的事件
upon event < pl, Deliver | q, [ACK, r] > such that r = rid do
    acks := acks + 1;  # 增加确认计数
    if acks > N/2 then
        acks := 0;  # 重置确认计数
        if reading = TRUE then
            reading := FALSE;  # 重置读取状态
            trigger < onar, ReadReturn | readval >;  # 完成读取操作并返回读取值
        else
            trigger < onar, WriteReturn >;  # 完成写入操作
```

这个算法的主要思想是：

- 每次写入或读取操作都会通过广播来更新所有进程中的值，确保它们是最新的。
- 读取操作不仅仅是返回本地的值，而是先通过广播确认当前的值，然后返回这个经过确认的值。
- 通过在所有正确的进程中广播并确认写入操作，算法确保即使在并发读写和进程崩溃的情况下，读取和写入操作的结果都能保持一致性和原子性。

正确性

1. **读操作的分析**
   - 假设 `rd1` 由进程 `pi` 发起，`rd2` 由进程 `pj` 发起。一旦 `rd1` 完成，就意味着 `pi` 已经向大多数进程写入了数据，因此大多数进程的时间戳至少为 `T1`。当 `pj` 执行 `rd2` 时，它也从大多数进程中读取数据。根据大多数原则，至少有一个进程会返回时间戳至少为 `T1` 的数据。因此，`pj` 后来发起的 `rd2` 不可能返回一个时间戳 `T2` 小于 `T1` 的值。这与我们的假设矛盾。
2. **性能**

- **写操作的通信步骤**：每次写操作需要两个通信步骤来完成。
- **读操作的通信步骤**：每次读操作需要四个通信步骤来完成。
- **消息复杂度**：每次操作的消息复杂度为O(N)，其中N是系统中的进程数。

多个写入者（MWMR）

当引入多个写入者的情况到原子寄存器中时，写入者们需要协调它们的时间戳以保持一致性和顺序。这里的核心挑战是确保所有写入操作都能被正确地排序，即使它们几乎同时发生。以下是实现多写单读（MWMR）原子寄存器的思路和规范：

1. **时间戳同步的思路**
   - **递增最高时间戳**：每个写入者在写入前，应该先确定所有已知写入的最高时间戳，并在此基础上递增时间戳。
   - **使用写入者ID作为时间戳的决断机制**：当两个写入操作具有相同的时间戳时，可以使用写入者的ID作为决断机制，确保写入操作的全局顺序。
2. **多写多读（MWMR）原子寄存器规范**
   - **原子性（Atomicity）**：
     - 每次读操作都返回在假定的执行中最近写入的值。
     - 每个失败的操作看起来要么是完成的，要么就好像从未被调用过。
     - 每个完成的操作看起来都在其调用和完成之间的某个瞬间执行。
3. **实现细节**
   - 在写入操作中，写入者首先从所有其他进程（或者一部分进程，如大多数进程）获取当前最高的时间戳，然后在此基础上递增自己的时间戳。
   - 写入操作除了包含要写入的值外，还需要包含时间戳和写入者的ID。
   - 读取操作需要从多个进程中收集值和时间戳，选择最高的时间戳对应的值作为读取结果。

实现

```
Implements: (N,N)-AtomicRegister, instance nnar.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl;
PerfectFailureDetector, instance P.

# 当原子寄存器初始化时触发的事件
upon event < nnar, Init > do
    (ts, wr, val) := (0, 0, ⊥);  # 初始化时间戳ts、写入者编号wr和寄存器值val
    correct := Π;  # 初始化correct集合，包含所有进程
    writeset := ∅;  # 初始化writeset集合，用于跟踪确认写入的进程
    readval := ⊥;  # 初始化读取值readval
    reading := FALSE;  # 初始化读取状态标志reading

# 当检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程

# 当有读取请求时触发的事件
```

```
upon event < nnar, Read > do
    reading := TRUE;  # 标记正在进行读取操作
    readval := val;  # 设置当前读取值为val
    trigger < beb, Broadcast | [WRITE, ts, wr, val] >;  # 广播当前值以确认最新状态

# 当有写入请求时触发的事件
upon event < nnar, Write | v > do
    trigger < beb, Broadcast | [WRITE, ts + 1, rank(self), v] >;  # 广播新的写入请求，递增时间戳并附上自身的rank

# 当通过尽力而为广播接收到写请求时触发的事件
upon event < beb, Deliver | p, [WRITE, ts', wr', v'] > do
    if ts'> ts ∨ (ts' = ts ∧ wr' > wr) then
        (ts, wr, val) := (ts', wr', v');  # 如果接收到的时间戳更大，或时间戳相同但写入者编号更大，则更新寄存器值
    trigger < pl, Send | p, [ACK] >;  # 向发送者p发送确认[ACK]

# 当通过点对点链接接收到确认时触发的事件
upon event < pl, Deliver | p, [ACK] > do
    writeset := writeset ∪ {p};  # 更新writeset集合

# 当所有正确的进程都确认了写入时触发的事件
upon correct ⊆ writeset do
    writeset := ∅;  # 重置writeset集合
    if reading = TRUE then
        reading := FALSE;  # 重置读取状态
        trigger < nnar, ReadReturn | readval >;  # 完成读取操作并返回读取值
    else
        trigger < nnar, WriteReturn >;  # 完成写入操作
```

这个算法的主要思想是：

- 每次写入操作都会通过广播来更新所有进程中的值，确保它们是最新的。
- 在写入操作中，时间戳被递增，且写入者的标识（如其进程编号或排名）被用作时间戳相同时的决断机制。
- 读取操作不仅仅是返回本地的值，而是先通过广播确认当前的值，然后返回这个经过确认的值。
- 通过在所有正确的进程中广播并确认写入操作，算法确保即使在并发读写和进程崩溃的情况下，读取和写入操作的结果都能保持一致性和原子性。