Linux High Availability & Clustering

Designing, Deploying, and Managing Fault-Tolerant Linux Infrastructure

Preface

Every minute of downtime costs money, erodes trust, and disrupts the people who depend on the services we build. In the world of Linux infrastructure, where organizations run everything from web applications to mission-critical databases, the ability to design systems that survive failure isn't a luxury—it's a fundamental expectation. This book was written to help you meet that expectation with confidence.

Why This Book Exists

Linux High Availability & Clustering was born from a simple observation: while there is no shortage of documentation on individual HA tools, there is a real gap when it comes to understanding how those tools fit together into a coherent, production-ready Linux infrastructure. Too many administrators learn clustering through trial and error—often during an outage, when the stakes are highest. This book aims to change that by providing a structured, end-to-end guide to designing, deploying, and managing fault-tolerant Linux systems.

Whether you are a Linux system administrator looking to build your first two-node cluster or an experienced engineer architecting multi-site failover across data centers, this book is designed to meet you where you are and take you further.

What You Will Learn

The journey begins with the foundational concepts of high availability—what it means, how it is measured, and why Linux has become the platform of choice for resilient infrastructure. From there, we dive deep into the core technologies that power Linux clustering: **Corosync** for cluster communication, **Pacemaker** for intelligent resource management, **DRBD** for real-time data replication, and the critical mechanisms of **fencing** and **quorum** that protect your data when things go wrong.

But understanding individual components is only part of the story. This book dedicates significant attention to *real-world application*—building highly available web services and databases on Linux, implementing load balancing and horizontal scaling, monitoring cluster health, and automating operations to reduce human error. The later chapters address multi-site high availability and the architectural thinking required to design production-ready Linux infrastructure that stands up to the demands of modern workloads.

How This Book Is Structured

The sixteen chapters are organized in a deliberate progression. **Chapters 1–4** establish the architectural foundation of Linux clustering. **Chapters 5–8** focus on storage, replication, and making critical services highly available. **Chapters 9–12** tackle the operational disciplines—fencing, quorum, monitoring, and automation—that separate a fragile cluster from a robust one. **Chapters 13–16** broaden the lens to multi-site designs, scalability, and the strategic mindset needed to evolve from system administrator to infrastructure architect.

The appendices provide practical, ready-to-use resources: a **Pacemaker command cheat sheet**, an **HA design checklist**, **fencing configuration and failover**

test plan templates, and a guide to building a career in Linux infrastructure. These are the tools I wish I had when I started this work.

Who This Book Is For

This book is for Linux professionals who refuse to accept "it just went down" as an answer. If you manage Linux servers and want to ensure they remain available through hardware failures, network partitions, and software crashes, you will find actionable knowledge on every page. A working familiarity with Linux system administration is assumed; a willingness to think critically about failure is essential.

Acknowledgments

No technical book is written in isolation. I am grateful to the open-source communities behind Corosync, Pacemaker, DRBD, and the countless Linux projects that make this work possible. Their dedication to building reliable, freely available software is the foundation upon which everything in these pages rests. I also owe a debt to the system administrators, site reliability engineers, and infrastructure architects whose real-world challenges and hard-won lessons shaped the practical focus of this book.

Downtime is inevitable. Extended downtime is a choice. Let's make sure your Linux infrastructure is ready for what comes next.

Bas van den Berg

Table of Contents

Chapter 1: Understanding High Availability Concepts

In the modern world of enterprise computing, downtime is not merely an inconvenience. It is a direct threat to revenue, reputation, and operational continuity. Every second that a critical system remains unavailable translates into lost transactions, frustrated users, and potentially catastrophic consequences for organizations that depend on their infrastructure to function around the clock. This reality has driven the evolution of a discipline known as High Availability, and Linux stands at the very center of this discipline as the dominant operating system powering the majority of mission-critical infrastructure worldwide.

This chapter lays the essential groundwork for everything that follows in this book. Before you configure a single cluster resource, write a heartbeat configuration file, or deploy a failover mechanism, you must first deeply understand the theoretical and practical foundations of High Availability. You must understand what it truly means, how it is measured, why it matters, and how the various components of a High Availability architecture work together to keep services running even when individual parts of the system fail. We will explore these concepts thoroughly, always through the lens of Linux, which provides the most robust, flexible, and cost-effective platform for building highly available systems.

What High Availability Really Means

High Availability, often abbreviated as HA, refers to the design approach and associated practices that ensure a system or service remains operational and accessible for a very high percentage of time. It is not about preventing failures entirely, because hardware will eventually fail, software will encounter bugs, and networks will experience disruptions. Instead, High Availability is about designing systems that can tolerate these failures gracefully, continuing to provide service to users with minimal or no perceivable interruption.

In the context of Linux, High Availability involves configuring multiple servers, often called nodes, to work together as a coordinated unit. When one node experiences a failure, another node detects this failure and assumes responsibility for the services that were running on the failed node. This process, known as failover, happens automatically and ideally completes so quickly that users are unaware anything went wrong.

It is important to distinguish High Availability from related but distinct concepts. High Availability is not the same as fault tolerance, although the two are often confused. A truly fault-tolerant system continues operating without any interruption whatsoever when a component fails, typically through hardware-level redundancy such as duplicate processors and memory modules operating in lockstep. High Availability, by contrast, acknowledges that a brief interruption may occur during failover but ensures that this interruption is measured in seconds rather than hours or days.

Similarly, High Availability is not the same as disaster recovery. Disaster recovery focuses on restoring services after a catastrophic event, such as a data center fire or a regional power outage, and typically involves restoring from backups or activating a geographically distant standby site. High Availability operates at a

more immediate level, handling the kinds of failures that occur within a single data center or across closely connected sites.

On Linux systems, the tools and technologies that enable High Availability have matured significantly over the past two decades. The Pacemaker cluster resource manager, the Corosync cluster communication system, DRBD for replicated block storage, and keepalived for virtual IP management are all open-source projects that run exclusively or primarily on Linux. These tools give Linux administrators the ability to build enterprise-grade High Availability solutions without the enormous licensing costs associated with proprietary alternatives.

Measuring Availability: The Language of Nines

To have meaningful conversations about High Availability, you need a precise way to measure and express it. The industry standard for measuring availability uses a metric expressed as a percentage of uptime over a given period, typically one year. This percentage is commonly referred to using the "nines" notation, which describes how many nines appear in the availability percentage.

The following table provides a comprehensive overview of the standard availability levels, their corresponding annual downtime, and the typical use cases and infrastructure requirements associated with each level.

Availability Level	Percentage	Annual Downtime	Monthly Downtime	Weekly Downtime	Typical Use Case	Infrastructure Requirement
One Nine	90.0%	36.53 days	73.05 hours	16.80 hours	Development and testing environments	Single server, no redundancy
Two Nines	99.0%	3.65 days	7.31 hours	1.68 hours	Internal business applications	Basic monitoring, manual failover
Three Nines	99.9%	8.77 hours	43.83 minutes	10.08 minutes	Standard commercial services	Redundant components, automated monitoring
Four Nines	99.99%	52.60 minutes	4.38 minutes	1.01 minutes	E-commerce, financial services	Fully redundant HA cluster with automatic failover
Five Nines	99.999%	5.26 minutes	26.30 seconds	6.05 seconds	Telecommunications, emergency services	Multi-site active-active clusters, no single point of failure
Six Nines	99.9999%	31.56 seconds	2.63 seconds	0.60 seconds	Critical national infrastructure	Extreme redundancy at every layer, custom engineering

Each additional nine represents a tenfold reduction in permissible downtime and a corresponding increase in the complexity and cost of the infrastructure required to achieve it. Moving from three nines to four nines on a Linux platform might involve transitioning from a simple active-standby pair of servers to a multi-node cluster with redundant networking, shared storage, and comprehensive monitoring. Moving from four nines to five nines typically requires eliminating every single point of failure in the entire stack, from power supplies and network switches to DNS resolution and storage controllers.

The availability percentage is calculated using the following formula:

```
Availability = (Total Time - Downtime) / Total Time * 100
```

For example, if a Linux web server cluster experienced a total of 4 hours of downtime over the course of a year, the availability would be calculated as follows:

```
Total minutes in a year = 365.25 * 24 * 60 = 525,960 minutes
Downtime in minutes = 4 * 60 = 240 minutes
Availability = (525,960 - 240) / 525,960 * 100 = 99.954%
```

This result falls between three nines and four nines, which means the system did not meet a four-nines target. Understanding these calculations is essential for setting realistic availability goals and for evaluating whether your Linux HA infrastructure is meeting its objectives.

Note: When calculating availability, it is critical to define clearly what constitutes "downtime." Planned maintenance windows are sometimes excluded from availability calculations in Service Level Agreements, but from the user's perspective, the service is still unavailable during those windows. A truly robust Linux HA architecture should allow for rolling upgrades and maintenance without any service interruption, making planned downtime a concept that can be largely eliminated.

Single Points of Failure: The Enemy of Availability

The single most important concept in High Availability design is the identification and elimination of single points of failure, commonly abbreviated as SPOF. A single point of failure is any component in your infrastructure whose failure would cause the entire service to become unavailable. If your web application runs on a single Linux server, that server is a single point of failure. If both of your clustered servers connect to the same network switch, that switch is a single point of failure. If your database cluster uses a single shared storage array, that storage array is a single point of failure.

Identifying single points of failure requires a systematic, layer-by-layer analysis of your entire infrastructure. Consider a typical Linux-based web application stack. The following table walks through each layer and identifies common single points of failure along with the standard HA mitigation strategy for each.

Infrastructure Layer	Component	Potential SPOF	Linux HA Mitigation
Power	Power supply unit	Single PSU in server	Dual PSUs connected to separate circuits, UPS systems
Network	Network interface	Single NIC	NIC bonding using Linux bonding driver or NetworkManager
Network	Switch	Single uplink switch	Redundant switches with bonded interfaces in 802.3ad mode

Network	Internet connection	Single ISP link	Multiple ISP connections with BGP or policy routing
Compute	Server hardware	Single server	Pacemaker/Corosync cluster with multiple nodes
Storage	Local disk	Single disk drive	Linux software RAID using mdadm, or hardware RAID
Storage	Storage system	Single storage array	DRBD replication, GlusterFS, or Ceph distributed storage
Application	Web server process	Single instance	Multiple instances behind HAProxy or keepalived with LVS
Application	Database	Single database server	MariaDB Galera Cluster, PostgreSQL streaming replication
DNS	Name resolution	Single DNS server	Multiple DNS servers, anycast DNS, round-robin records

The process of eliminating single points of failure is sometimes called "redundancy engineering," and it follows a straightforward principle: every critical component must have at least one backup that can take over its function when the primary fails. On Linux, this principle is implemented at every layer of the stack using a rich ecosystem of open-source tools.

For example, Linux NIC bonding allows you to combine multiple physical network interfaces into a single logical interface that continues to function even if one of the physical interfaces fails. You can configure this directly through the Linux kernel's bonding module. A basic bonding configuration might look like this:

```
# Load the bonding kernel module
modprobe bonding

# Verify the module is loaded
lsmod | grep bonding

# Create a bond interface configuration
cat > /etc/sysconfig/network-scripts/ifcfg-bond0 << EOF
DEVICE=bond0
TYPE=Bond
BONDING_MASTER=yes
IPADDR=192.168.1.100
NETMASK=255.255.255.0
GATEWAY=192.168.1.1
ONBOOT=yes
BOOTPROTO=none
BONDING_OPTS="mode=active-backup miimon=100 primary=eth0"
EOF
```

In this configuration, the `mode=active-backup` parameter tells the bonding driver to use one interface as the primary and automatically switch to the backup interface if the primary fails. The `miimon=100` parameter sets the link monitoring interval to 100 milliseconds, ensuring that a failed link is detected within a fraction of a second.

Note: The bonding modes available in the Linux kernel include balance-rr (mode 0), active-backup (mode 1), balance-xor (mode 2), broadcast (mode 3), 802.3ad (mode 4), balance-tlb (mode 5), and balance-alb (mode 6). For High Availability purposes, active-backup and 802.3ad are the most commonly used modes. The 802.3ad mode requires switch support for Link Aggregation Control Protocol (LACP) but provides both redundancy and increased throughput.

The Anatomy of a Linux HA Cluster

A Linux High Availability cluster is composed of several key components that work together to detect failures and respond to them automatically. Understanding these components and their roles is essential before you begin building your own clusters.

The first component is the **cluster communication layer**. This is the foundation upon which everything else is built. The cluster communication layer is responsible for allowing the nodes in the cluster to communicate with each other, to determine which nodes are currently alive and healthy, and to agree on the current state of the cluster. In modern Linux HA clusters, this role is filled by Corosync, which uses a protocol called Totem to maintain a reliable, ordered communication ring among all cluster nodes. Corosync sends heartbeat messages between nodes at regular intervals, and if a node fails to respond within a configured timeout, it is declared dead by the remaining nodes.

The second component is the **cluster resource manager**. This is the brain of the cluster, responsible for deciding which resources should run on which nodes and for orchestrating failover when a node fails. In the Linux ecosystem, Pacemaker is the dominant cluster resource manager. Pacemaker maintains a model of the cluster's desired state, which includes definitions of all the resources (such as IP addresses, filesystems, and application services), the constraints that govern where and how those resources should run, and the rules that determine what should happen when failures occur. When Pacemaker detects that the actual state of the cluster differs from the desired state, it takes corrective action automatically.

The third component is the **resource agents**. These are scripts or programs that Pacemaker uses to manage individual resources. A resource agent knows how to start, stop, and monitor a specific type of resource. For example, there is a resource agent for managing an Apache web server, another for managing a virtual

IP address, another for managing a filesystem mount, and so on. Linux provides hundreds of resource agents through the resource-agents package, covering everything from simple services to complex database systems. Resource agents follow the Open Cluster Framework (OCF) specification, which defines a standard interface that Pacemaker uses to interact with them.

The fourth component is the **fencing mechanism**, also known as STONITH, which stands for "Shoot The Other Node In The Head." Fencing is perhaps the most misunderstood component of a Linux HA cluster, but it is absolutely critical. When a node becomes unresponsive, the remaining nodes cannot be certain whether the unresponsive node has truly crashed or whether it is simply experiencing a temporary communication problem. If the remaining nodes start the failed node's resources without being certain that the failed node has stopped them, both nodes might try to run the same resources simultaneously. For a database, this could result in catastrophic data corruption. Fencing solves this problem by forcibly shutting down or isolating the unresponsive node before its resources are started elsewhere. On Linux, fencing is typically implemented through IPMI commands that power off the failed node, through management interfaces on virtual machines, or through network-based power switches.

The following table summarizes these core components:

Component	Purpose	Linux Implementation	Role in Failover
Cluster Communication	Node discovery, heartbeat, membership	Corosync	Detects node failures through missed heartbeats
Cluster Resource Manager	Resource placement, failover decisions	Pacemaker	Decides where to move resources after failure

Resource Agents	Start, stop, and monitor individual resources	OCF scripts, systemd agents, LSB scripts	Executes the actual start and stop operations
Fencing / STONITH	Ensure failed nodes are truly offline	fence_ipmilan, fence_virsh, fence_aws	Powers off failed node before resource recovery
Quorum	Prevent split-brain scenarios	Votequorum (part of Corosync)	Determines which partition can continue operating

The fifth component listed in this table, **quorum**, deserves special attention. In a cluster with multiple nodes, it is possible for a network failure to divide the cluster into two or more groups of nodes that can communicate within their group but not with other groups. This situation is called a "split brain," and it is extremely dangerous because each group might believe that the other group has failed and attempt to take over its resources. Quorum is the mechanism that prevents this. A group of nodes has quorum if it contains more than half of the total number of nodes in the cluster. Only the group that has quorum is allowed to continue operating and managing resources. The other group, lacking quorum, must stop all resources and wait for communication to be restored.

Practical Exercise: Evaluating Your Current Infrastructure

To solidify your understanding of the concepts covered in this chapter, perform the following exercise on a Linux system that you manage or have access to.

First, identify all the services running on the system that would be affected by a server failure:

```
# List all active services
```

```
systemctl list-units --type=service --state=running

# Identify listening network services
ss -tlnp

# Check for mounted network filesystems
mount | grep -E 'nfs|cifs|gluster'

# Review the system's network configuration for redundancy
ip link show
cat /proc/net/bonding/bond0 2>/dev/null || echo "No bonding
configured"
```

Second, document the current availability characteristics of the system:

```
# Check system uptime
uptime

# Review recent reboot history
last reboot | head -20

# Check for any disk redundancy
cat /proc/mdstat 2>/dev/null || echo "No software RAID
configured"
lsblk -f

# Examine power supply status if available through IPMI
ipmitool sdr type "Power Supply" 2>/dev/null || echo "IPMI not
available"
```

Third, create a simple availability report by examining the system logs for any recent service interruptions:

```
# Search for recent service failures in the journal
journalctl --since "30 days ago" --priority=err --no-pager | head
-50

# Check for any OOM (Out of Memory) kills
journalctl --since "30 days ago" | grep -i "out of memory" | wc
-l
```

```
# Review kernel messages for hardware errors
dmesg | grep -iE "error|fail|fault" | tail -20
```

After running these commands, create a document that lists every single point of failure you can identify in the system. For each SPOF, note what Linux HA technology could be used to mitigate it. This exercise will give you a concrete starting point for the cluster implementations we will build in subsequent chapters.

Note: The commands above use standard Linux utilities that are available on virtually all distributions. The `systemctl` command is specific to systems using systemd as their init system, which includes all major modern Linux distributions such as Red Hat Enterprise Linux, CentOS, Ubuntu, Debian, SUSE Linux Enterprise, and Fedora. If you are working with an older system that uses SysVinit, replace `systemctl list-units` with `service --status-all`.

Moving Forward

The concepts presented in this chapter form the intellectual foundation for everything else in this book. High Availability is not simply a collection of tools and configurations. It is a design philosophy that requires you to think systematically about failure, to anticipate what can go wrong, and to build systems that respond to failure automatically and gracefully. Linux provides an extraordinarily powerful platform for implementing this philosophy, with a mature ecosystem of clustering tools that rival and often surpass their proprietary counterparts in both capability and flexibility.

As you move into the next chapter, you will begin translating these concepts into concrete implementations. You will install and configure Corosync and Pacemaker on real Linux systems, create your first cluster, and experience firsthand the process of automated failover. The theoretical understanding you have gained

here will make those practical exercises far more meaningful, because you will understand not just what the tools are doing, but why they are doing it and how each piece fits into the larger picture of a robust, fault-tolerant Linux infrastructure.

Remember that achieving High Availability is an iterative process. You do not need to eliminate every single point of failure on day one. Start by identifying the most critical services and the most likely failure scenarios, address those first, and then progressively improve your infrastructure over time. Each step you take toward eliminating single points of failure brings you closer to the level of availability your organization requires, and Linux gives you every tool you need to get there.

Chapter 2: Linux Clustering Architecture

Understanding the architecture behind Linux clustering is not merely an academic exercise. It is the foundation upon which every reliable, fault-tolerant production system is built. When a critical database server fails at two in the morning and your cluster seamlessly redirects traffic to a standby node without a single user noticing, that is the architecture doing its job. When that same failure causes a cascading outage that takes down your entire e-commerce platform, that is the architecture failing. The difference between these two outcomes lies entirely in how well you understand, design, and implement your clustering architecture on Linux.

This chapter takes you deep into the structural components, communication models, resource management strategies, and decision-making algorithms that form the backbone of every Linux cluster. We will examine each layer of the architecture, from the physical network interconnects to the abstract resource agents that manage your services, and we will do so with the practical depth that real-world deployments demand.

The Fundamental Layers of a Linux Cluster

A Linux cluster is not a single piece of software. It is a carefully orchestrated stack of components, each responsible for a distinct function, and each depending on the others to maintain the illusion of a single, always-available system. Think of it as a

layered cake where removing any single layer causes the entire structure to collapse.

At the lowest level, you have the **infrastructure layer**, which consists of the physical or virtual machines running Linux, the network interfaces connecting them, and the shared or replicated storage they access. Above that sits the **messaging and membership layer**, responsible for allowing nodes to communicate with each other and agree on which nodes are currently alive and participating in the cluster. The next layer up is the **resource management layer**, which decides where services should run and what to do when something goes wrong. Finally, at the top, you have the **resource agent layer**, which contains the scripts and programs that actually start, stop, and monitor individual services like databases, web servers, and file systems.

The following table provides a comprehensive overview of these layers and their responsibilities.

Layer	Primary Responsibility	Key Linux Components	Failure Impact
Infrastructure	Physical connectivity, compute, storage	Linux kernel, NIC drivers, multipath, LVM	Total cluster failure if not redundant
Messaging and Membership	Node communication, quorum determination	Corosync, Kronosnet, UDP/UDPU transports	Split-brain, data corruption
Resource Management	Service placement, failover decisions	Pacemaker (CRM), policy engine	Services not restarted or misplaced
Resource Agents	Service control (start, stop, monitor)	OCF scripts, systemd agents, LSB scripts	Individual service failure

Each of these layers deserves careful attention, and we will explore them all in the sections that follow.

The Infrastructure Layer: Building the Physical Foundation

Every Linux cluster begins with its infrastructure. The nodes themselves are typically servers running a mainstream Linux distribution such as Red Hat Enterprise Linux, SUSE Linux Enterprise Server, Debian, or Ubuntu Server. The choice of distribution matters because each provides different levels of integration with clustering software, different kernel versions, and different support lifecycles.

Network connectivity between cluster nodes is arguably the most critical infrastructure decision you will make. Cluster nodes must communicate constantly, exchanging heartbeat messages, synchronizing state, and coordinating resource management. If this communication is interrupted, the cluster cannot distinguish between a failed node and a network partition, leading to the dreaded split-brain scenario where both nodes believe they are the sole survivor and attempt to run the same services simultaneously.

For this reason, production Linux clusters should always use **redundant network paths** for cluster communication. This is typically achieved through network bonding at the Linux kernel level, combined with physically separate network switches.

To configure a bonded interface for cluster communication on a Linux system, you would create a configuration similar to the following using NetworkManager:

```
nmcli connection add type bond con-name cluster-bond ifname bond0
\
    bond.options "mode=active-backup,miimon=100,primary=eth1"

nmcli connection add type ethernet con-name bond-slave-1 ifname
eth1 \
    master bond0

nmcli connection add type ethernet con-name bond-slave-2 ifname
eth2 \
```

```
    master bond0

nmcli connection modify cluster-bond ipv4.addresses 10.10.10.1/24
\
    ipv4.method manual

nmcli connection up cluster-bond
```

The `mode=active-backup` option ensures that one interface is always active while the other stands ready to take over immediately if the primary fails. The `mi-imon=100` parameter tells the kernel to check the link status of each interface every 100 milliseconds, providing rapid detection of network cable failures or switch port issues.

Note: The cluster communication network should be a dedicated, isolated network that carries no application traffic. Mixing cluster heartbeat traffic with application data introduces the risk that a burst of application traffic could delay heartbeat messages, causing the cluster to falsely declare a node dead.

Storage architecture in a Linux cluster falls into two broad categories: shared storage and replicated storage. Shared storage means that multiple nodes can access the same physical or logical storage device, typically through a Storage Area Network using Fibre Channel or iSCSI. Replicated storage means that each node has its own local storage, and data is synchronized between nodes in real time using software such as DRBD (Distributed Replicated Block Device).

The following table compares these two approaches:

Characteristic	Shared Storage (SAN/iSCSI)	Replicated Storage (DRBD)
Cost	Higher (requires SAN infrastructure)	Lower (uses local disks)
Performance	Generally higher throughput	Write performance reduced by replication
Complexity	Requires SAN administration expertise	Requires DRBD configuration and tuning

Data copies	Single copy on shared device	Two or more copies across nodes
Network dependency	Separate storage network required	Replication uses cluster network
Scalability	Scales well with SAN expansion	Limited to DRBD node count
Fencing integration	SCSI reservations available	Requires separate fencing mechanism

The Messaging and Membership Layer: Corosync in Depth

Corosync is the heartbeat of nearly every modern Linux cluster. It implements the Totem Single-Ring Ordering and Membership protocol, which provides reliable, ordered message delivery to all nodes in the cluster. When Pacemaker needs to tell all nodes about a configuration change or a resource state transition, it sends that message through Corosync, which guarantees that every node receives the message in the same order.

The Corosync configuration file lives at `/etc/corosync/corosync.conf` and defines the fundamental behavior of the cluster communication layer. Here is a production-quality configuration for a two-node cluster:

```
totem {
    version: 2
    cluster_name: production-cluster
    transport: knet

    crypto_cipher: aes256
    crypto_hash: sha256

    interface {
```

```
        linknumber: 0
        knet_transport: udp
    }

    interface {
        linknumber: 1
        knet_transport: udp
    }
}

nodelist {
    node {
        ring0_addr: 10.10.10.1
        ring1_addr: 10.10.20.1
        name: node1
        nodeid: 1
    }

    node {
        ring0_addr: 10.10.10.2
        ring1_addr: 10.10.20.2
        name: node2
        nodeid: 2
    }
}

quorum {
    provider: corosync_votequorum
    two_node: 1
}

logging {
    to_logfile: yes
    logfile: /var/log/cluster/corosync.log
    to_syslog: yes
    timestamp: on
}
```

Let us examine the critical elements of this configuration. The `transport: knet` directive tells Corosync to use the Kronosnet transport layer, which is the modern