UDC 004.75

A. Lavrov, M. Lytvynenko,
O. Syrota, P. Rodionov A. Humeniuk

SERVICE DISCOVERY AT SCALE: LIMITATIONS OF CENTRALIZED MODELS AND DECENTRALIZED ALTERNATIVES

Abstract: Service discovery is a critical component in cloud-native and microservice-based architecture, enabling dynamic service registration and resolution. While centralized service discovery systems such as Consul, Eureka, and Kubernetes DNS are widely adopted, they introduce scalability limitations, single points of failure, and operational complexity when deployed at scale. This paper investigates the inherent challenges of centralized service discovery models in large-scale, distributed environments and evaluates decentralized alternatives. The aim of the research is to show how decentralized service discovery model can improve system availability and make a large-scale distributed system more resilient and resistant to failure.

Keywords: service discovery, distributed systems, cloud-native architecture, microservices, gossip protocol, decentralized discovery, scalability, fault tolerance.

Introduction

In today's technological landscape, enterprise-grade systems are increasingly designed to be cloud-native to meet the performance, scalability, and resilience demands of modern workloads. Well-architected cloud-native systems typically adhere to a defined set of principles such as those outlined in the Reactive Manifesto[1].

Within modern software engineering, several architectural paradigms have emerged to support such systems, including Service-Oriented Architecture (SOA), Microservices Architecture, and Event-Driven Architecture (EDA). Despite their differences, all these paradigms share a common requirement: independent components must be able to communicate effectively. Some architectures emphasize asynchronous, message-driven interactions, while others are built around synchronous request – response models.

However, regardless of the interaction style, one foundational capability is essential: service discovery.

“Service discovery the ability for components to dynamically locate and connect to one another in a fluid, often ephemeral environment.” – *Sam Newman, “Building Microservices”, O’Reilly, 2015*[2].

Several patterns exist for implementing service discovery, including:

 ISSN 1560-8956

Client-side service discovery

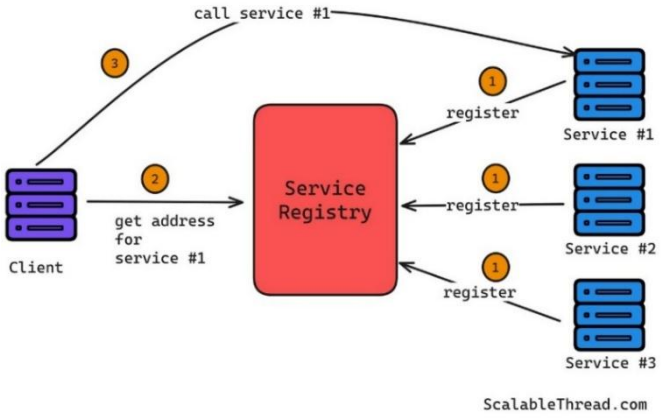

Figure 1. Client-side service discovery [3]

Client-side service discovery[4] is a *decentralized service resolution pattern* in which the service consumer is responsible for performing **service instance resolution** and **endpoint selection**. Upon invocation, the client queries a **service registry –** a distributed or centralized repository that maintains the metadata (e.g., network addresses, health status, capabilities) of active service instances. The client then applies a **selection strategy** (e.g., round-robin, least connections, latency-aware) to determine the target instance and establish a direct communication channel.

In this pattern, the registry functions as a **passive directory service**, while the **discovery logic and load-balancing responsibility** reside entirely within the client. This reduces intermediate network hops but increases coupling between the client and the service registry API.

Server-side (proxy-based) service discovery

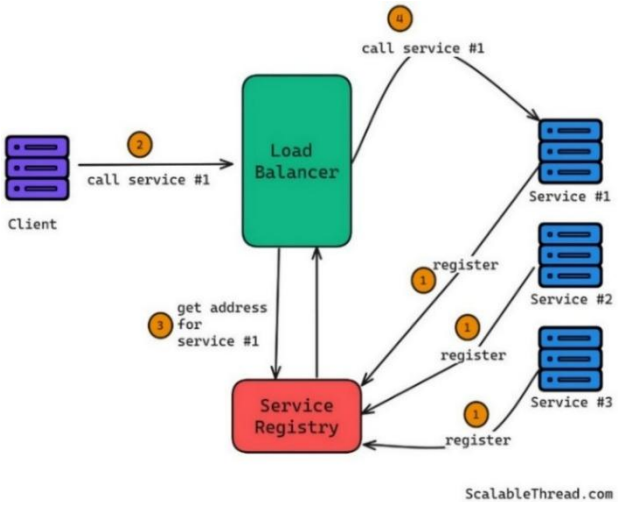

Figure 2. Server-side service discovery [3]

Server-side service discovery[5] is a *centralized service resolution pattern* in which the service consumer sends a request to a **fixed network endpoint** (often a load balancer, gateway,

or API proxy) without performing direct service instance resolution. The intermediary component queries the **service registry** to determine the set of healthy service instances and performs **request routing** or **load balancing** on behalf of the client.

In this pattern, the **discovery logic and routing decisions** are abstracted away from the client, reducing its complexity and eliminating the need for registry integration in client code. However, it introduces an additional network hop and potential single points of failure if the routing layer is not highly available.

While the service discovery approaches are effective in many scenarios, they all share a fundamental architectural limitation: centralization. Whether through a DNS-based mechanism, a dedicated service registry (such as Eureka or Consul), these systems often introduce single points of failure and scalability bottlenecks. Such limitations become particularly problematic in large-scale or multi-region deployments, where high availability and fault tolerance are critical.

Centralized Models: Strengths and Limitations

At this part we will dig deeper into centralized models of service discovery and analyse them more thoughtfully. Before moving forward, it is important to clearly define the difference between centralized and decentralized models. A centralized model can, in fact, be physically distributed across multiple nodes or datacenters. This approach can make sense in terms of operational efficiency and load distribution. However, even when physically distributed, such systems remain **logically centralized** because they still require a single, consistent view of state, achieved through coordination mechanisms like consensus (RAFT[6], PAXOX[7]). Therefore, in this article, *centralization* should be understood as a **logical** property – referring to reliance on a single, authoritative control plane – rather than a purely physical deployment characteristic.

Benefits of Centralized Service Discovery

Despite their limitations, centralized service discovery systems provide several practical advantages that have contributed to their widespread adoption in production environments. Firstly, they offer a single source of truth for service metadata, allowing all clients and infrastructure components to resolve service locations consistently. Consistency in one of the most important characteristics that we have considered when design a distributed system. (CAP theorem[8]). In addition, centralization simplifies observability, auditing, and management by providing a unified control point for service registration, health status, and configuration.

Additionally, centralized discovery models often support fine-grained access control, version-aware routing, and service tagging, all of which are critical in complex microservice environments where behavioral control and policy enforcement are required. These benefits make centralized service discovery an attractive and pragmatic choice for many enterprises

 ISSN 1560-8956

and cloud-native architectures, particularly where consistency, predictability, and operational control are priorities.

Problem statement

While centralized service discovery systems such as Consul, Kubernetes DNS (CoreDNS + etcd), and similar registry-based solutions have proven effective in moderate-scale deployments, their architectural characteristics introduce significant challenges in large-scale or globally distributed environments. Those systems create inherent limitations:

1. Single Points of Failure and Bottlenecks[9] – Centralized components can become both a performance bottleneck and a critical dependency whose failure affects the availability of the entire service discovery layer.
2. Scaling Constraints Under High Churn[10] – In high-velocity environments where services are frequently created, destroyed, or relocated, write throughput limits, leader election delays, and quorum dependencies introduce latency and instability.
3. Operational Overhead – Maintaining low-latency, highly available centralized registries demands extensive capacity planning, observability, and tuning, increasing the operational burden on platform engineering teams.

Even hybrid solutions like Consul Connect, which combine centralized service registration with decentralized failure detection, ultimately depend on a consistent central control plane that inherits the same scaling and availability constraints. In Kubernetes, DNS-based discovery depends on CoreDNS and etcd, both of which exhibit performance degradation and resolution failures under extreme load [11].

These limitations suggest that as systems grow – particularly in multi-region, multi-availability-zone topologies – the cost and complexity of sustaining centralized service discovery at scale becomes prohibitive, motivating the need to investigate decentralized or hybrid approaches that can better tolerate failures, reduce bottlenecks, and adapt to dynamic workloads.

This problem can also be analyzed through the lens of reliability engineering theory. A distributed system can be represented as a set of N nodes, each characterized by its individual reliability *Ri(t)*. When all nodes depend on a single centralized component, the overall system behaves as a **series system**. In such a configuration, the failure of any single component inevitably leads to the failure of the entire system. The reliability of a series system is expressed as:

$$R_{SERIAL}(t) = R_1(t) \cdot R_2(t) \cdot R_3(t) = \prod_{i=1}^{n} R_i(t)$$

In contrast, when the nodes are organized in a **parallel system**, the system remains operational as long as at least one component continues to function. The reliability of a parallel configuration is defined by:

$$R_{PARALLEL}(t) = 1 - \left[\left(1 - R_1(t)\right)\left(1 - R_2(t)\right)\left(1 - R_3(t)\right)\right]$$

Or more generally:

$$R_{PARALLEL}(t) = 1 - \prod_{i=1}^{n}\left(1 - R_i(t)\right)$$

To illustrate this with a practical example, assume that the availability of a single machine in a service discovery cluster is 0.99. For a mid-size cluster consisting of 10 machines, the reliability under a series system is:

$$0.99^{10} \approx 0.904$$

This implies that even with highly reliable individual nodes, the cumulative reliability of a centralized (series) model decreases substantially as the cluster size grows.

By comparison, in a parallel system of the same 10 machines, the reliability is:

$$R_{PARALLEL} = 1 - (0.01)^{10} \approx 0.9999$$

This near-perfect reliability demonstrates the advantage of decentralized or replicated approaches, where the system continues to function even if a significant subset of nodes fails.

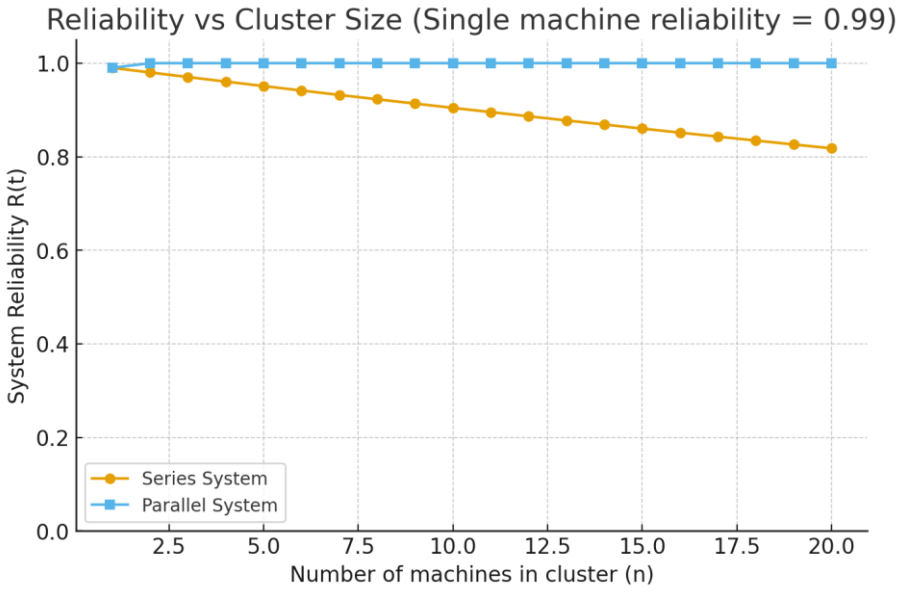

Figure 3. Series System vs Parallel system comparison

Decentralized Alternatives

Decentralization, on the other hand, is a concept that has become increasingly popular in recent years. There are many reasons for this shift, but in our context, the most relevant are **system availability** and **fault tolerance**. In modern cloud computing, component failures are not exceptional – they are expected. We must embrace this reality and design systems with resilience as a core principle. A good reference point here is the **chaos engineering** culture, which promotes proactively testing systems against failure scenarios to ensure they can withstand and recover from them.

ISSN 1560-8956

The most important and distinguishing characteristic between **distributed** and **decentralized** systems is how resources and responsibilities are allocated. In a decentralized system, resources must be **necessarily spread across all participating processes or nodes**, ensuring that no single entity holds exclusive control. In a distributed system, by contrast, resources only need to be **sufficiently** spread to meet operational requirements, which may still leave certain components as critical dependencies.

This distinction makes decentralized systems inherently more resilient to single points of failure. In a truly decentralized architecture, the workload is necessarily shared among all members, eliminating reliance on any individual node. A clear example can be seen by comparing **Bitcoin** and **Visa**:

- Visa, while highly distributed, still operates through centralized infrastructure. A system outage – though infrequent – can disrupt payment processing globally, posing a risk to transaction availability and security.
- Bitcoin, as a decentralized ledger, has no single control point. The probability of the entire system failing is significantly lower.

Of course, decentralized peer-to-peer systems like Bitcoin are not without their own challenges – for instance, vulnerabilities such as the **51% attack** – but these issues fall outside the scope of this research.

Of course, while gaining the following benefits, there is a price to pay — in our case, that price is consistency[12]. As we have already stated, service discovery is the backbone of any distributed, service-oriented system, and in this context, availability is far more important than strict consistency. More precisely, we trade a strong consistency model for eventual consistency to achieve higher availability and tolerance to network-partitioning[13].

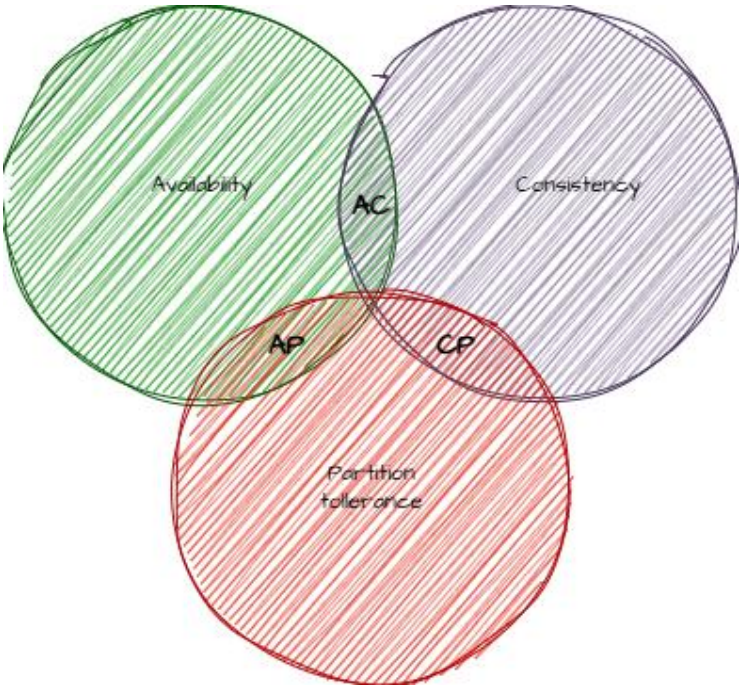

Figure 4. CAP theorem visualization

An additional point supporting this approach is that in large-scale distributed systems, the environment changes so rapidly that maintaining service discovery as eventually consistent and ephemeral is often not a significant problem.

Considering the growing importance of cloud platforms in business operations, more companies are paying closer attention to decentralization. This article does not advocate for building *only* fully decentralized systems. In practice, decentralized architectures can be complex, difficult to design, and challenging to debug and maintain.

Instead, our goal is to take **proven concepts from decentralization –** particularly those that enhance availability and fault tolerance – and apply them within well-established distributed system paradigms. By selectively integrating these principles, organizations can improve resilience without incurring the full complexity and operational overhead of a purely decentralized design.

Possible solutions and research directions

As an alternative, the following paper proposes a decentralized design with no central coordinator. Discovery is removed as a standalone service and pushed into domain-oriented services. Each service periodically exchanges state with randomly selected peers, allowing knowledge of the overall system to converge over time. This can be realized with a gossip-based (epidemic) protocol [14].

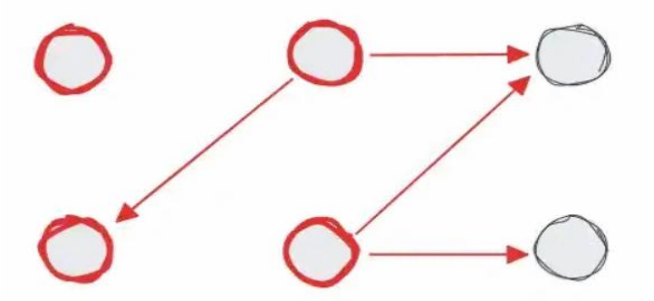

Figure 5. Gossip communication between nodes

The gossip protocol is a decentralized peer-to-peer communication mechanism designed to distribute information efficiently across large-scale distributed systems. Its core principle is that each node periodically exchanges messages with a randomly selected subset of other nodes. Through these repeated interactions, the system eventually achieves message dissemination with high probability. In simpler terms, the gossip protocol allows nodes to develop a global view of the system through many small, localized exchanges.

All nodes communicate with each other using the UDP protocol. UDP is the logical choice in this context because it is connectionless, eliminating the need for session establishment and teardown as required by TCP. This reduces protocol overhead and avoids the additional latency introduced by handshakes. As a result, UDP enables lightweight, low-latency message exchange while minimizing the amount of traffic propagated through the network.

 ISSN 1560-8956

When discussing fully distributed systems, it is essential to consider the initial state of a node as it joins a cluster. A newly initialized node must be able to discover at least a minimal set of peers – typically two or more active nodes – to begin participating in the dissemination process. Hardcoding such addresses in configuration files is impractical, as cloud environments are inherently dynamic and subject to frequent changes over time. Instead, an effective solution can be drawn from existing peer-to-peer (P2P) systems, such as BitTorrent, using seed nodes.

In a decentralized network, seed nodes function as stable entry points that facilitate the onboarding of new participants. They provide an initial "directory" of reachable peers, enabling new nodes to bootstrap their connectivity, synchronize with the network, and subsequently join the gossip-based dissemination process. Once integrated, the new node can rely entirely on peer-to-peer interactions, thus preserving the decentralized nature of the system while ensuring efficient and fault-tolerant network membership discovery.

Although automatic peer discovery using protocols such as ARP may seem appealing, it is limited to a single broadcast domain and is unsuitable for multi-subnet or cloud-based environments where network segmentation and virtual networking are common. Additionally, ARP-based discovery introduces unnecessary broadcast traffic and lacks built-in mechanisms for validating the liveness or role of discovered peers. In contrast, a seed node setup provides a scalable, controlled, and cloud-friendly approach that avoids these drawbacks while maintaining predictable and secure cluster formation.

Main formula for the number of seed nodes:

$$S(N) = max(3, \lceil ln(N) \rceil + 1)$$

where:

3 – the minimum number of nodes for redundancy,

N – the total number of nodes in the cluster,

2 – an additional offset used when $N \geq 1$.

Per Availability Zone or physically separated cluster:

$$S(N)_{\text{a}}z = max(3, \lceil (ln(N) + 1)/Az \rceil)$$

All of the concepts outlined above are viable but not trivial to implement. Moreover, the proposed approach should aim to reduce operational overhead while making the associated benefits easier to realize. Gossip algorithm implementation can be complex and should ideally be abstracted away from the compute node itself. Since gossiping is primarily an infrastructure-level concern rather than application logic, it can be implemented using a sidecar pattern. In this model, each compute node runs alongside a dedicated sidecar container responsible for gossip-based communication and client-side load balancing. This separation streamlines the node's core responsibilities while enabling resilient and scalable service discovery.

The proposed solution also functions as a proxy, routing requests to the appropriate dedicated node based on the current state of the internal gossip membership list. It can support various load-balancing algorithms, allowing requests to be directed to the destination host

using a strategy best suited for the specific use case. Considering eventual consistency of the system, it is reasonable to expect occasional delays or discrepancies in the membership list. For example, a specific Node Y of Service Z might be down, but this information has not yet been updated in the list. In such cases, traffic can be rerouted to another available instance of Node Y. By following this implementation approach, the target service remains decoupled and unaware of infrastructure details, simplifying deployment and cleanly separating business logic from cross-cutting concerns.

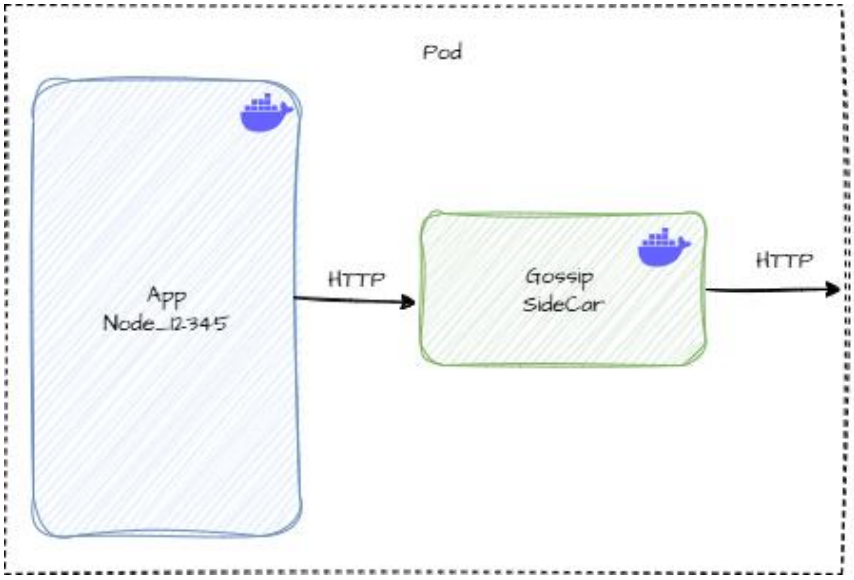

Figure 6. Sidecar deployment approach

Conclusion and Future Work

This work examined the critical role of service discovery as a foundational component in modern distributed, service-oriented systems. The discussion highlighted its importance for ensuring reliable communication and scalability, while also identifying the potential risks associated with logically centralized discovery models. By exposing these limitations, the study emphasized the need for alternative approaches that enhance availability, resilience and reduce operational fragility.

Building on these findings, future research will adopt a more practical perspective, focusing on the design and implementation of a service discovery mechanism that preserves the identified architectural benefits while minimizing operational overhead. This will include developing a working prototype, refining implementation details, and ensuring seamless integration into existing distributed environments. We will also address the network-flooding behavior inherent in epidemic communication, adding advanced techniques to reduce overhead.

Further work will also involve conducting performance evaluations and comparative analyses between centralized and decentralized discovery models in real-world, cloud-based deployments.

 ISSN 1560-8956

REFERENCES

1. The Reactive Manifesto – Available from: https://www.reactivemanifesto.org/
2. Sam Newman. Building Microservices 2 Edition / Sam Newman // Building Microservices 2 Edition. 30 April 2021. – P. 157-174.
3. What is Service Discovery – Available from: https://newsletter.scalablethread.com/p/what-is-service-discovery
4. Pattern: Client-side service discovery – Available from: https://microservices.io/patterns/client-side-discovery.html
5. Pattern: Server-side service discovery – Available from: https://microservices.io/patterns/server-side-discovery.html
6. *D. Ongaro, J. Ousterhout*. In search of an understandable consensus algorithm [Electronic resource] / D. Ongaro, J. Ousterhout // USENIX ATC'14: Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, 19-20 June 2014. – Philadelphia, PA, USA, 2014. – Available from: https://dl.acm.org/doi/10.5555/2643634.2643666
7. Paxos Made Simple – Available from: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf
8. *S. Gilbert, N. Lynch*. Perspectives on the CAP Theorem [Electronic resource] / S. Gilbert, N. Lynch // Computer. – 2012. – Vol. 45, iss. 2 – P. 30-36. – Available from: https://doi.org/10.1109/MC.2011.389
9. Amazon EKS announces native support for autoscaling CoreDNS Pods – Available from: https://aws.amazon.com/about-aws/whats-new/2024/05/amazon-eks-native-support-autoscaling-coredns-pods/
10. Service discovery at Stripe – Available from: https://stripe.com/blog/service-discovery-at-stripe
11. https://dl.acm.org/doi/abs/10.1145/3284028.3284034#core-history
12. "The art of service discovery at scale" by Nitesh Kant – Available from: https://www.youtube.com/watch?v=27ynM2tbNXM&t=983s
13. Airbnb Service Discovery: Past, Present, Future (Challenges of Change) – Chase Childers, Airbnb – Available from: https://www.youtube.com/watch?v=XQjOhJtw1wg&t=2023s
14. *S. Boyd, A. Ghosh, B. Prabhakar, D. Shah*. Gossip algorithms: design, analysis and applications [Electronic resource] / S. Boyd, A. Ghosh, B. Prabhakar, D. Shah // Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005. – Miami, FL, USA, 2005. – Available from: https://doi.org/10.1109/INFCOM.2005.1498447