Unstructured P2P network에서의 Hybrid 노드 탐색 알고리즘

사이버보안학과 202220664

허한빈

프로젝트 개요

peer-to-peer (P2P) network는 각 노드가 클라이언트와 서버의 역할을 모두 수행하는 분산 시스템이다. P2P 네트워크의 주요 장점 중 하나는 확장 가능한 아키텍처로, 새로운 노드가 네트워크에 연결될 때마다 시스템의 총 용량이 증가한다. P2P 네트워크의 이러한 측면은 특정 애플리케이션에 필요한 모든 크기로 확장 가능하다.

P2P network을 진행하기 위해선 우선적으로 노드 Discovery 단계가 선행되어야 한다. 데이터가 한 곳에 저장되어 있는 client-server 아키텍처와 달리 P2P 네트워크에서는 데이터가 peer 간에 분산되어 있다. 따라서 파일을 찾고 검색하는 행위가 매우 까다롭다. 특히 unstructured network 상에서 난이도가 급증한다. discovery 문제를 해결하기 위해 flooding, random walk, Random Walk with Neighbors Table, Normalized Flooding (NF), 등 다양한 기존 방법이 사용되지만 모두 장단점이 명확하다. Flooding은 빠른 탐색이 가능하지만, 자원과 대역폭이 많이 소모되는 단점이 있다. 반면에 랜덤 워크는 대역폭 소모가 적지만, 원하는 노드를 찾는 데 시간이 오래 걸릴 수 있다. 따라서 이번 프로젝트에서 unstructured P2P 네트워크 상에서 복합적 노드 Discovery 방식을 탐구할 것이다.

기존 알고리즘 동작 원리

- Flooding: unstructured P2P 네트워크에서 가장 많이 사용되는 리소스 검색 기법으로, discovery를 하고자 하는 peer가 가까운 모든 이웃 peer에게 query를 보낸다. 그 이후 각 peer는 query를 보낸 peer를 제외한 모든 이웃 peer에게 검색 query를 전달한다. 일종의 무제어 포트 배정이라고 할 수 있다. 검색 프로세스는 리소스가 발견되거나 질문이 TTL이 0이 될 때까지 계속된다. 모든 query는 초기 TTL(Time-to-live) 값으로 시작하며, query가 두 노드 사이를 이동할 때 TTL 값은 1씩 감소한다. 한 peer가 동일한 query를 여러 번 수신하는 문제를 방지하기 위해 각 query에는 고유 번호가 부여된다. Peer가 동일한 번호의 query를 이미 받았다면, 해당 query는 버려진다.

- Random Walk: Flooding 검색의 과도한 리소스 사용을 감소시키기 위해 제안되었다. 이 알고리즘에서 peer는 query를 전달하지 않고 선택한 이웃 중 일부에게 무작위로 모든 이웃에게 query를 전달하는 대신 무작위로 선택한 일부 이웃에게 query를 전달하여 검색을 시작한다. 파일이 발견되지 않으면 프로세스가 계속 진행되며, 각 peer는 하나 이상의 이웃을 무작위로 선택하여 query를 전달한다. 랜덤 워크에서 각 검색 query는 Flooding과 마찬가지로 TTL 값을 갖는다. query TTL은 노드 사이를 이동할 때마다 감소하며, query의 TTL 값이 0이 아닌 한 query는 전달된다.

- Random Walk with Neighbors Table: 무작위 탐색에서 각 peer는 이웃 peer가 소유한 파일 목록이 포함된 테이블을 가지고 있다. 검색 프로세스가 시작되면 검색을 시작한 peer는 해당 테이블에서 요청된 리소스에 대한 항목을 찾는다. 테이블에 이웃 peer 중 하나에 해당 리소스가 있는 것으로 표시되면 해당 peer로 검색이 진행된다. 그렇지 않으면 기존의 랜덤 워크 알고리즘과 유사한 방식으로 검색이 수행된다.

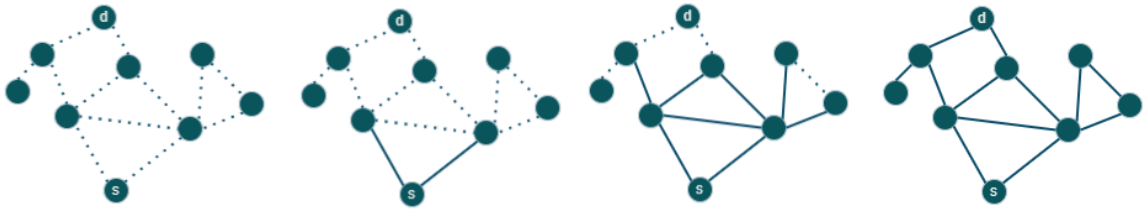


Figure 1 Flooding 노드 detection example

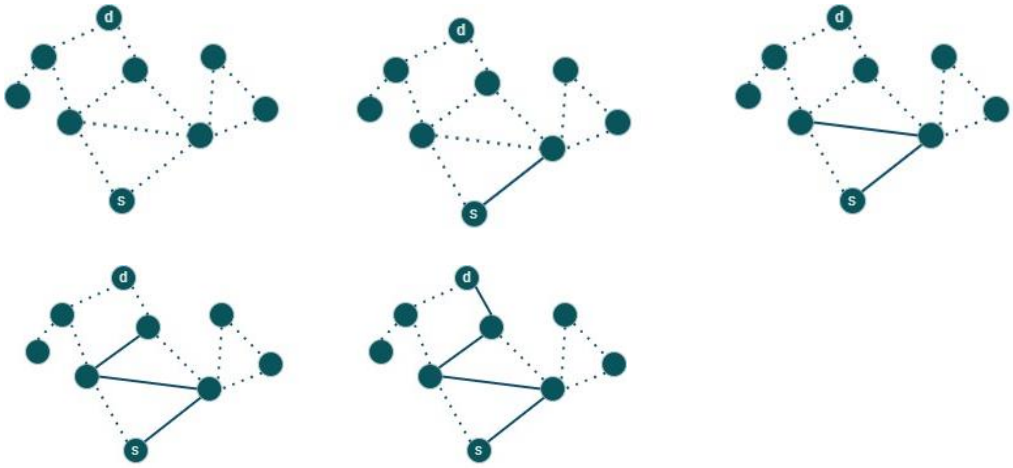


Figure 2 Random walk 노드 detection example

새 알고리즘

Flooding의 핵심적인 문제점 중 하나는 목표 노드(destination; d)와 시작 노드 (start; s)의 거리가 근본적으로 먼 경우에 있다. 해당 케이스에서 모든 노드는 필요치 않은 리소스를 감당해야만 한다. 최악의 경우 영역 내에 있는 모든 노드를 탐색하고 나서야 목표 노드를 발견할 수 있을 것이다. 여기서 거리는 한 peer에서 다른 peer로 가기 위해서 써야 하는 자원의 양을 말한다.

이러한 자원 낭비를 해결하기 위해 'random walk를 통한 end point의 탐지와 router의 flooding 방식'을 제안한다.

1. Start 노드에서의 Random walk 수행
2. Query를 중복되지 않고 수신할 수 없는 상태의 노드인 endpoint의 발견
   => 이를 기점으로 discovery를 random walk에서 flooding으로 전환.
3. endpoint에게 query를 수신한 노드, router를 기점으로 한 flooding 알고리즘 수행

노드 A가 노드 B에 도달할 수 있는 유일한 경로일 경우를 가정해보자. 노드 A는 유일하게 B와 연결되어 있는 상태이며 B를 통해 query를 전달받았다. 이는 child 역할을 하는 end deivce A와 parent 역할을 하는 라우터 B로 볼 수 있다. 라우터는 네트워크 기기의 패킷을 저장하고 있으며 언제든지 전달할 수 있는 상태를 유지하는 기기를 말한다. 최종 기기 (end device)는 단일 라우터만 통신하며 다른 네트워크 기기의 패킷을 저장 또는 전달하지 않는다.

만약 노드 B가 연결된 모든 경로에서 query가 방문했다면, B는 해당 query를 송신할 노드 경로가 더이상 존재하지 않는다. figure 3, figure 4 케이스를 포함해 query를 보낼 수 없는 상태의 모든 노드를 end point라 명시한다.

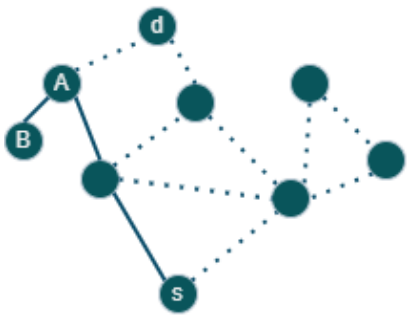


Figure 3 B에 도달할 수 있는 유일한 노드 A

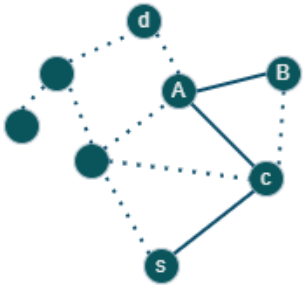
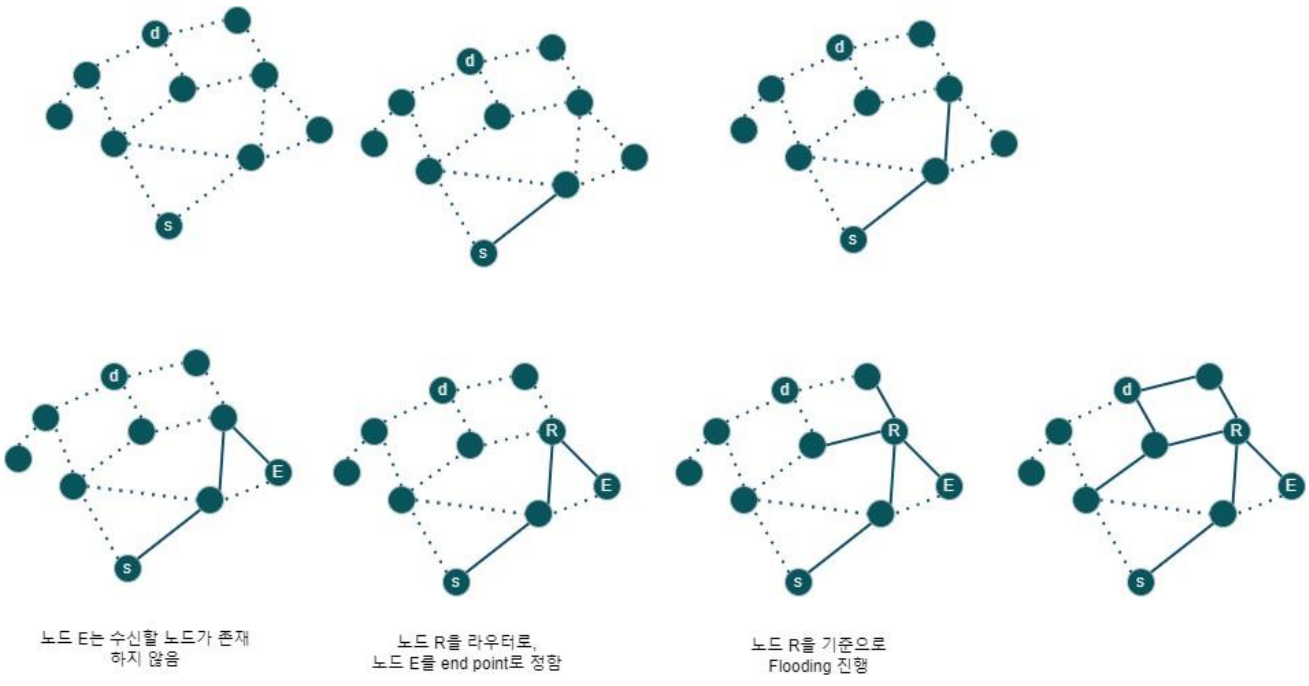


Figure 4 B가 수신할 노드가 없는 경우

해당 노드가 end point임을 알아내는 방법은 여러 가지가 있다. 먼저 figure 3의 경우 B가 query를 A에게 수신했을 때 A는 이미 query의 고유 번호와 B에게 수신한 로그 기록이 스택에 저장되어 있는 상태다. 따라서 query를 보낸 노드와 query를 받은 노드가 동일하다면 노드 B는 라우터다. Figure 4의 경우 Random walk로 진행될 때 B는 A에게, 또는 C에게 보낼 수 있다. A로 query를 보낼 경우 figure3와 동일한 케이스임으로 end point임을 탐지할 수 있다. C에게 보낼 경우 노드 C는 스택에 이미 query의 고유 번호가 저장되어 있으므로, B에게 해당 query를 이미 받았다는 ACK를 수신한다. 따라서 연결된 모든 노드에 수신하였으나 모두 ACK를 받았을 경우 해당 노드는 end point 이다. 또한 해당 노드 B에게 query를 전달한 노드 A를 라우터 라 한다.

End point와 router를 탐지한 이후 flooding 방식을 수행한다. 기존 방식과 동일하게 destination 노드에서 리소스가 발견되거나 query의 TTL이 0이 될 때까지 반복한다



구현

네트워크 환경을 비슷하게 구현하기 위해 구성하는 단계에서 우선 연결, join & leave 노드 개념을 도입시켰다.

```python
import networkx as nx
import random

#global 변수
m = 2  # 최소 degree
μ = 0.1  # 노드가 네트워크를 떠날 확률
N = 0  # 기존 네트워크의 초기 노드 ID
G = nx.Graph()  # 기존 네트워크 그래프
Ntarget = 100  # 목표 노드수


# 네트워크 생성(노드 추가, 우선연결)
def grow(Ntarget, τj, τl):
    global N
    for i in range(m+1, Ntarget):
        join(N, τj)
        num = random.random()
        if N == Ntarget:
            break
        if num < μ:
            Ndel = random.randint(1, N)
            leave(Ndel, τl)

#G2의 노드를 사용하여 G1에 우선 연결을 수행.
def preferential_attachment(G1, G2):
    #성공한 새 링크 수를 반환
    new_links = 0
    for node in G2:
        if node in G1:
            G1.add_edge(random.choice(list(G1.nodes)), node)
            new_links += 1
    return new_links

# Nrand의 이웃 노드를 포함한 부분 그래프를 최대 hop 거리로 가져옴.
def get_subgraph(Nrand, hops):
    # TTL 결정 판단 기준
    return nx.ego_graph(G, Nrand, radius=hops)
```

```python
def join(i, τj):
    global N
    N += 1
    numoflinks = 0
    while numoflinks < m:
        Nrand = random.randint(1, N)
        myG = get_subgraph(Nrand, τj)
        numoflinks += preferential_attachment(G, myG)

def leave(i, τl):
    myG = get_subgraph(i, τl)
    G.remove_node(i)
    preferential_attachment(G, myG)

# Random walk 알고리즘
def random_walk(node):
    visited = set()
    while True:
        neighbors = list(G.neighbors(node))
        if not neighbors:
            break
        next_node = random.choice(neighbors)
        if next_node in visited:
            return node
        visited.add(next_node)
        node = next_node
    return node

#Flooding 알고리즘
def flooding(node):
    queue = [node]
    visited = set()
    while queue:
        current = queue.pop(0)
        if current in visited:
            continue
        visited.add(current)
        for neighbor in G.neighbors(current):
            if neighbor not in visited:
                queue.append(neighbor)
    return visited
```

```python
# eedpoint 판별
def is_endpoint(node):
    neighbors = list(G.neighbors(node))
    for neighbor in neighbors:
        if not set(G.neighbors(neighbor)).issubset(neighbors + [node]):
            return False
    return True

# endpoint 발견 기점으로 알고리즘 전환
def perform_discovery(start_node):
    endpoint = random_walk(start_node)
    if is_endpoint(endpoint):
        flooding(endpoint)
#τj = 조인 프로세스 중에 하위 그래프에 포함할 무작위로 선택된 노드에서 떨어진 홉 수
# 연결할 노드를 찾기 위해 우선 연결 프로세스가 어느 정도까지 진행되는지
#τi = 탈퇴 프로세스 동안 하위 그래프에서 고려할 네트워크를 떠나는 노드에서 떨어진 홉 수
#나가는 노드에 연결된 노드를 다시 연결하기 위해 네트워크가 얼마나 멀리 떨어져야 하는지

# Example Usage
if __name__ == "__main__":
  grow(Ntarget, τj=2, τl=2)
  start_node = random.randint(1, N)
  perform_discovery(start_node)
```

수학적 분석

Random walk는 TTL이 만료되거나 리소스를 찾을 때까지 쿼리를 무작위로 선택된 이웃에게 전송한다. k는 초기에 제공된 TTL 값이나, 리소스를 발견했을 때의 TTL이다.

$$\text{After } k \text{ steps, some } R(k) = d \cdot (d-1)^{k-1}$$

Figure 5 Random walk 성능

$$S = \sum_{k=1}^{N} k \cdot \mathbb{P}[k] = \sum_{k=1}^{N} k \cdot \frac{r}{N}(1-\frac{r}{N})^{k-1} \approx N/r \text{ for } 1 \ll r \leq N.$$

Figure 6 Flooding 성능

$$T_{random\_walk} = O(N_{visited})$$

$$T_{is\_endpoint} = O(d)$$

$$T_{flooding} = O(|V|+|E|)$$

End point가 탐지될 때까지 랜덤 워크를 수행한다. 네트워크 구조에 따라 복잡성이 달라지기도 하지만 평균적으로는 노드 수에 비례한다 (end point Detection) 노드가 end point인지 확인하는 것은 노드의 이웃을 조사함으로 결정할 수 잇다. 이는 O(d)로 근사할 수 있으며, 여기서 d는 노드의 degree이다. 러딩은 기점에서 재귀적으로 도달 가능한 모든 이웃 노드를 방문한다. 최악의 경우 도달할 수 있는 전체 하위 그래프를 방문한다. BFS (Breadth-First Search)

|V| = 노드의 수, |E| = 서브 그래프 내의 edge의 수, TTL: Time-to-live value for the queries

응용 가능성 및 다른 시나리오

Random Walk with Neighbors Table의 경우 각 peer은 이웃 peer가 소유한 파일 목록이 포함된 테이블을 가지고 있다. Random walk를 진행 중에 메시지 패킷을 가지고 있는 노드가 임의로 라우터가 되는 한 노드를 정해주고, 패킷을 전송한다. 이후 패킷을 가지고 있는 라우터를 포함하여 하나의 hop 그룹을 생성한다. 해당 라우터는 1 hop 안에서 flooding 방식을 수행한다. 라우터가 hop 그룹 안에 있는 모든 링크를 가지고 있는 일종의 superpeer 역할을 수행하는 것이다.

만약 그 hop 그룹 안에 존재하지 않는다면 이전 hop 그룹에 있는 노드들을 제외한 새로운 hop 그룹을 생성한다. 라우터는 그 전 그룹에서 라우터와 물리적으로 가장 먼 거리에 존재하는 노드로 한다.

한 노드가 라우터를 임의로 선정하는 방법에 대하여 블록체인의 지분 증명과 유사하게 진행한다. 각 노드가 사용 가능한 메모리 자원의 크기에 따라 라우터를 선정한 확률을 결정한다. 라우터로 선정된 노드는 discovery를 수행할 역할로서 CPU와 메모리 일부를 할당한다. 작업이 끝난 후, 즉 메시지 패킷과 목표 노드 주소 데이터의 이전이 끝난 후 모든 메모리를 삭제하고, 자원을 다시 회수한다.

주안점

가장 핵심으로 보고 있는 부분은 'random walk와 flooding 방식의 전환 시점'이다. 기존 연구 논문을 찾아봤을 때 dynamic, 또는 hybrid 방식으로 새 알고리즘을 제안한 논문이 많았다. router 개념을 도입하여 기점을 선정하여 기존 연구과의 구별점을 만들어냈다. 해당 모델은 network 안에 있는 노드의 개수가 많을수록, edge 개수가 적을수록 더 작은 시간 복잡도와 더 높은 성능을 가져올 수 있다.

기대 효과

- 효율적인 리소스 사용: 더 많은 메모리를 가진 노드를 라우터로 사용함으로써 알고리즘은 쿼리 패킷을 효율적으로 처리하고 전체 네트워크 트래픽을 줄일 수 있다.

- Dynamic Adaptation: 알고리즘은 동적으로 적응하여 현재 네트워크 토폴로지 및 리소스 가용성을 기반으로 새로운 라우터를 선택한다.

- Targeted Flooding: 홉 네트워크 그룹 내의 Flooding은 관련 노드 수를 최소화하여 중복성과 리소스 소비를 줄인다.

취약점 및 개선점

이 프로젝트에서 제시한 방식은 '노드 간의 거리가 먼 경우'을 상정하고 제안하였다. 따라서 목표 노드가 노드 중앙에 있거나 라우터의 정반대에 위치해 있다면 기존 flooding 방식과 비슷하거나 더 낮은 효과를 낼 수 있다. 이러한 케이스들을 분석하고 범용적인 알고리즘으로 발전시킬 필요가 있다.

실제 장비와 기술적 구현의 제한적 상황으로 인해 실제 환경에서 query data를 주고 받지 못하

였다. 추후 몇 개의 노드들을 실제로 로컬 네트워크 상에 연결하여 시도할 예정이다.

네트워크 정보가 없을 때 네트워크 상의 주소를 이용하여 목적지까지 경로를 체계적으로 결정하기 위해 random walk와 flooding 방식을 상황에 맞춰 수행하는 것이 가장 리소스를 절약하고 시간을 최소화할 수 있는 방법이다.

References

Laila Bashmal ·Asma Almulifi ·Heba Kurdi. Hybrid Resource Discovery Algorithms for Unstructured Peer-to-Peer Networks. The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)

Durgesh Rani Kumari ·Hasan Guclu ·Murat Yuksel. Ad-hoc limited scale-free models for unstructuredpeer-to-peer networks. Peer-to-Peer Netw. Appl. (2011) 4:92–105

MAARTEN VAN STEEN ANDREW S. TANENBAUM. DISTRIBUTED SYSTEMS 4th edition