AFLL: MMO 게임 서버의 순환 인과관계 학습 기반 실시간 부하 안정화

강신석[1], 김영재†
[1]서강대학교 AI · SW대학원
kang.shinsuk@gmail.com, youkim@sogang.ac.kr

AFLL: Real-time Load Stabilization for MMO Game Servers Based on Circular Causality Learning

Shinsuk Kang[1], Youngjae Kim†
[1]Graduate School of AI · SW, Sogang University, Seoul

요 약

대규모 다중 사용자 온라인(MMO) 게임 서버는 수천 명의 동시 접속자가 생성하는 높은 메시지 처리 부하로 인해 성능 저하를 경험한다. 서버가 전송하는 메시지가 클라이언트의 추가 요청을 유발하는 순환 인과관계(circular causality)가 피드백 루프를 형성하여 부하를 급격히 증폭시킨다. 본 논문은 역전파(backpropagation)를 활용한 적응적 피드백 루프 학습(AFLL, Adaptive Feedback Loop Learning) 시스템을 제안한다. AFLL은 각 메시지 타입이 부하에 미치는 영향을 실시간 추적하고, 경사하강법으로 전송률을 동적 조정하여 부하 급증을 사전 방지한다. 1,000명 동시 접속자 실험 결과, AFLL은 학습 비활성화 대비 47.5%의 CPU 시간 감소(5,851ms → 3,070ms), 65.5%의 스레드 경합 감소(63.03% → 21.76%), 성능 스파이크 완전 제거를 달성했다. 학습 OFF와 학습 ON 모두 탁월한 재현성(변동계수 0.55%와 3.85%)을 보였으며, 세 번의 학습 ON 실험 모두 동일 가중치로 수렴하였다.

1. 서론

MMO(Massively Multiplayer Online) 게임 서버는 수많은 사용자가 동시에 접속하여 상호작용하는 고부하 실시간 시스템이다. 본 논문에서 플레이어(player)는 게임에 참여하는 사용자를 의미하며, 클라이언트(client)는 플레이어가 사용하는 게임 클라이언트 프로그램을 지칭한다. MMO 게임에서는 일관된 게임 상태 유지를 위해 각 플레이어의 위치, 행동, 상태 변화가 모든 주변 플레이어에게 실시간으로 동기화되어야 한다. 플레이어 수 (n)이 증가하면 전송해야 할 상태 정보량이 $n \cdot (n-1)$로 비선형적으로 급증한다. 100명이 밀집된 지역에서는 $100 \times (100-1) = 9,900$건의 상태 전송이 필요하며, 1,000명이 참여하는 대규모 전투에서는 약 100만 건으로 폭증한다. 이러한 부하 급증으로 서버가 처리 한계에 가까워지면 응답 지연 급증, 게임 공정성 붕괴, 사용자 이탈이 발생하여 서비스 지속 가능성을 위협한다 [2].

이 문제의 핵심은 서버가 전송한 메시지가 다시 플레이어의 추가 요청을 유발하는 **순환 인과관계**(circular causality) 때문이다. 이는 일반적인 웹 서버의 선형적 부하 증가와 근본적으로 다르다. 예를 들어, 서버가 플레이어에게 주변 10명의 정보를 전송하면, 플레이어는 이들(= 주변 10명)과의 상호작용을 위해 서버로 추가 요청을 보낸다. 결과적으로 "전송→요청→부하→지연→재요청"의 피드백 루프가 형성되어 부하가 기하급수적으로 증폭된다. 더 심각한 것은 메시지 타입마다 부하 기여도가 다르다는 점이다. 플레이어 사망 알림은 단발성이지만, 주변 플레이어 위치 정보는 지속적인 추적 요청을 유발한다. 기존 정적 쓰로틀링이나 반응적 부하 분산 기법은 이러한 동적 피드백 구조를 학습하지 못하여 근본적 해결이 어렵다.

기존 연구의 한계. 단일 서버 부하 안정화를 위한 기존 연구들은 크게 두 가지 접근으로 구분된다. (1) *전송 효율화*: Smart Reckoning [4]은 머신러닝으로 플레이어 움직임을 예측하여 불필요한 위치 업데이트 전송을 줄이는 방식으로, 78.76%의 예측 정확도를 달성했으나 고정된 로직으로 동작하여 런타임 중 부하 변화에 적응하지 못한다. (2) *처리 효율화*: LEARS [3]는 락프리 상태 모델로 처리량을 향상시켰으나, 요청 폭증 자체를 억제하지 못해 응답 지연과 스레드 경합이 급증한다. 결국 기존 연구는 부하 "결과"에만 대응했을 뿐, 순환 인과관계라는 근본 원인을 학습하거나 제어하지 못했다.

학습 기반 접근의 필요성과 기술적 장벽. 메시지 타입별 부하 기여도는 상황에 따라 동적으로 변하므로, 학습을 통해 이를 실시간 추적하고 임계점 도달 전에 선제 제어하면 전송량 감소와 처리 부담 경감이 동시에 달성된다. 그러나 기존 연구는 사전에 설정된 고정 로직이나 정적 규칙을 사용해왔다. 학습 알고리즘을 매 전송 결정마다 수행하면 수백 마이크로초의 지연을 발생시켜 초당 수만 건 메시지 처리 환경에서 치명적이기 때문이다. 결과적으로 학습의 필요성을 인식하면서도 오버헤드 부담으로 정적 정책에 의존할 수밖에 없었다.

본 연구의 접근과 기여. 본 연구는 역전파 [1] 기반 적응적 피드백 루프 학습(AFLL, Adaptive Feedback Loop Learning)을 제안하여 위 장벽을 극복한다. AFLL의 핵심 차별점은 세 가지이다: **첫째**, 역전파로 서버 부하 예측 오차를 최소화하면서 메시지 타입별 부하 기여도를 동적 추적한다. **둘째**, 학습과 제어를 병렬 스레드로 분리하여 0ms 오버헤드를 달성함으로써 학습 기법의 성능 부담 문제를 해결한다. **셋째**, 부하 발생 후 대응하는 반응적 접근이 아닌, 순환 인과관계를 학습하여 부하 급증 전에 선제 억제하는 *사전적 부하 제어* 패러다임을 제시한다. 이러한 역전파 기반 피드백 루프를 통한 부하 예측 오차 최소화 접근은 기존의 경험적 조정이나 정적 규칙과 본질적으로 차별화되는 새로운 방법이다. 실험 결과, 1,000명 동시 접속자 환경에서 47.5% CPU 감소와 성능 스파이크 완전 제거를 달성하였다.

2. AFLL 시스템 설계

표기법. 본 연구는 6가지 메시지 타입을 정의하였다: DEATH(사망), DAMAGE(피해량), PROJECTILE(투사체), CONE(범위공격), OWN_STATE(자신위치), NEARBY_PLAYERS(주변위치). i는 메시지 타입 인덱스 ($i \in [1,6]$), w_i는 타입 i의 학습된 가중치, $c_i(t)$는 시점 t에 전송된 타입 i 메시지 개수, $L(t)$는 부하 점수 ($\in [0,1]$), ΔL_{pred}는 예측된 부하 변화, ΔL_{actual}은 실제 관측

∗ 본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단 (RS-2025-00564249)의 지원을 받아 수행되었다.
† Corresponding Author

된 부하 변화, $\alpha = 0.03$은 학습률(빠른 수렴), $\beta = 0.9$는 모멘텀 계수(부하 변동 잡음 평활화)이다.

2.1 시스템 아키텍처

AFLL은 각 메시지 타입이 서버 부하에 기여하는 정도를 학습하여 부하 급증을 사전 방지하며, 학습과 제어를 두 개의 병렬 스레드로 분리하여 0ms 오버헤드를 달성한다. *메인 스레드*는 학습된 가중치 w_i로 메시지 전송 여부를 실시간 결정하고(2.3절), *백그라운드 스레드*는 1초마다 전송 기록을 분석하여 역전파로 가중치를 업데이트한다(2.2절). 두 스레드는 공유 메모리를 통해 협력하며, 메인 스레드는 학습을 수행하지 않고 가중치 조회만 하므로 메시지 처리 경로에 오버헤드가 없다.

2.2 역전파 학습 프로세스

백그라운드 스레드는 1초마다 공유 큐에 축적된 전송 기록을 읽어 역전파로 가중치를 학습한다. 학습은 순방향 전파와 역방향 전파 두 단계로 이루어진다. 먼저 지난 1초간 타입별 전송량 $c_i(t)$를 집계하고, 현재 가중치 w_i로 부하 변화를 예측한다:

$$\Delta L_{\text{pred}}(t) = \sum_{i=1}^{6} w_i \cdot c_i(t)$$

1초 후 실제 부하 변화 $\Delta L_{\text{actual}}(t)$를 관측하고, 예측 오차를 손실 함수로 정의한다. 1초 윈도우는 순환 인과관계의 피드백 루프(서버 전송→클라이언트 처리→추가 요청→서버 처리, 평균 400-500ms)를 포함하며, 본 연구는 서버가 제어 가능한 출력 메시지만을 학습 대상으로 한다(클라이언트 입력은 서버가 직접 조절할 수 없고, 부하의 주요 원인은 n^2 복잡도의 상태 브로드캐스트이기 때문이다):

$$\mathcal{L}(t) = \frac{1}{2}(\Delta L_{\text{actual}}(t) - \Delta L_{\text{pred}}(t))^2$$

그래디언트를 계산하고 모멘텀을 포함한 경사하강법으로 가중치를 업데이트한다:

$$\frac{\partial \mathcal{L}}{\partial w_i} = -(\Delta L_{\text{actual}} - \Delta L_{\text{pred}}) \cdot c_i(t)$$

알고리즘 1: 역전파 학습 (백그라운드, 1초마다)

```
function backpropagation():
  // 순방향: 예측 계산
  ΔL_pred = 0
  for each msgType i in [1..6]:
    ΔL_pred += w[i] * countsSent[i]

  // 역방향: 그래디언트 및 업데이트
  ΔL_actual = L(t) - L(t-1)
  error = ΔL_actual - ΔL_pred
  for each msgType i in [1..6]:
    grad = -error * countsSent[i]
    velocity[i] = β*velocity[i] - α*grad
    w[i] = w[i] + velocity[i]
    w[i] = clamp(w[i], minW[i], maxW[i])
```

2.3 학습된 가중치의 활용

메인 스레드는 메시지 전송 요청마다 학습된 가중치 w_i를 조회하여 전송 여부를 실시간 결정한다. 알고리즘 2는 4단계 의사결정 과정을 보여준다. 핵심 설계 원칙은 (1) 게임플레이 필수 메시지(DEATH, DAMAGE)는 무조건 전송하고, (2) 나머지 메시지는 가중치에 비례하여 전송 빈도를 동적 조절한다는 것이다.

알고리즘 2: 메시지 전송 결정 (메인, 0ms 오버헤드)

```
function shouldTransmit(msgType, L(t)):
  // Step 0: 중요 메시지 보호
  if msgType in {DEATH, DAMAGE}:
    return TRUE

  // Step 1: 최소 FPS 보장 (여유 임계값)
  if timeSince >= minInterval[msgType]
     and L(t) < 0.85:
    return TRUE

  // Step 2: 예측 기반 허용 (목표 임계값)
  w = getWeight(msgType)
  L_pred = L(t) + w * 1.0
  if L_pred <= 0.70:
    return TRUE

  // Step 3: 확률적 차단
  over = (L_pred - 0.70) / 0.30
  P_block = min(w × over × 1.2
                + queuePenalty,
                maxRate[msgType])
  return random() > P_block
```

Step 0은 게임플레이 필수 정보인 DEATH와 DAMAGE를 무조건 전송하여 플레이어 경험을 보장한다. **Step 1**과 **Step 2**는 서로 다른 두 단계의 임계값을 사용한다: Step 1은 *여유 임계값*(0.85)으로 과부하가 아닌 경우 최소 FPS를 보장하기 위한 주기적 전송을 허용하며, Step 2는 더 엄격한 *목표 임계값*(0.70)으로 예측 부하 $L_{\text{pred}} = L(t) + w_i$가 목표치 이하일 때만 전송을 허용한다. 이 이중 임계값 설계는 부하가 0.70-0.85 구간에서는 최소 FPS는 보장하되 새로운 전송은 제한하여 부하 상승을 억제하는 완충 구간을 형성한다. **Step 3**은 목표치 초과 시 확률적 차단을 수행한다. 먼저 초과 정도를 정규화한다: over $= (L_{\text{pred}} - 0.70)/0.30$은 목표치(0.70)부터 최대치(1.0)까지의 구간(0.30)에서 현재 초과 정도를 0-1 범위로 정규화한 값이다. 차단 확률은 $P_{\text{block}} = \min(w_i \times \text{over} \times 1.2 + \text{queuePenalty}, \text{maxRate}[i])$로 계산되며, 여기서 $w_i \times$ over는 가중치와 초과 정도를 곱하여 부하 기여도가 높은 메시지일수록 더 강하게 차단하고, 1.2 증폭 계수는 빠른 억제를 위한 조정값이며, queuePenalty는 큐 크기에 따른 추가 패널티, maxRate는 타입별 최대 차단률 상한을 의미한다.

특히, OWN_STATE와 NEARBY_PLAYERS(각각 $w = 0.949$)는 밀집 상황에서 주변 다수에게 위치 정보를 브로드캐스트하여 부하를 기하급수적으로 증폭시키는 핵심 원인이다. 이들은 가장 높은 가중치를 학습하므로 부하가 높아질수록 전송 빈도와 범위(주변 몇 명까지 전송할지)가 동적으로 축소되며, PROJECTILE($w = 0.699$) 등 낮은 가중치 타입은 상대적으로 덜 차단된다. 이처럼 AFLL은 가중치에 비례한 타입별 차등 제어로 밀집 상황의 증폭 효과를 사전 차단한다.

2.4 시스템 최적화

알고리즘 1-2를 직접 적용하면 심각한 성능 문제가 발생한다: 예측 계산 125µs, 통계 수집이 200개 윈도우(20초) 순회로 O(n×200) 복잡도 2,847µs, 역전파 학습 1,523µs가 소요된다. 본 연구는 세 단계 최적화로 해결하였다: (1) **예측 캐싱**은 가중치 업데이트 시에만 재계산하여 125µs→8µs 단축, (2) **증분 통계 캐시**는 O(n×200)→O(1) 개선으로 2,847µs→12µs 단축, (3) **백그라운드 학습**은 학습을 별도 스레드 분리로 메시지 처리 경로를 0ms 달성했다. 전체 4,495µs→20µs(99.6% 단축)를 달성했다.

125µs, 2,847µs, 1,523µs 등은 Java 프로파일링 실측값

3. 실험 평가

실험은 LEARS [3]의 락프리 상태 모델 위에 AFLL을 적용한 MMO 게임 서버 프로토타입(Java 17)에서 수행되었다. 1,000명의 시뮬레이션 클라이언트가 이동 및 공격 행동을 확률적으로 수행하여 실제 게임 플레이를 재현하였으며, Intel Core i7-12700K(12코어), 32GB RAM 환경에서 각 30분씩 실험하였다. 목표 서버 부하는 0.70으로 설정하였다.

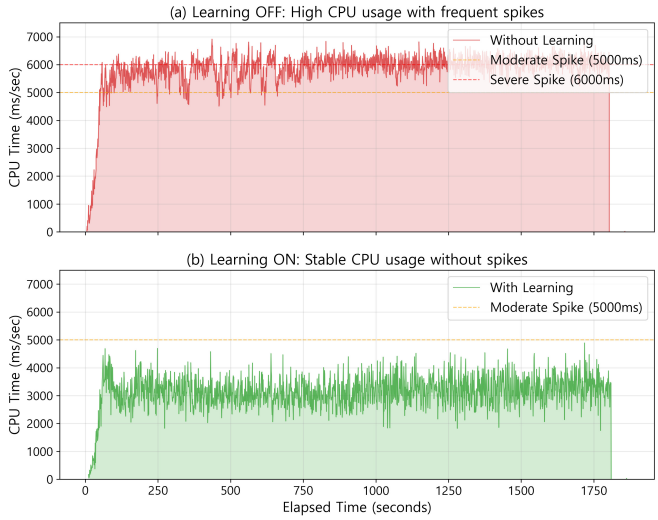

그림 1: CPU 시간 시계열 비교 (30분)

그림 1은 30분간의 CPU 시간 변화를 보여주며, 학습 OFF는 큰 폭의 스파이크가 빈번하게 발생하는 반면 학습 ON은 안정적인 패턴을 유지한다.

표 1: CPU 성능 및 스레드 경합 통계 비교

메트릭	학습 OFF*	학습 ON*	개선
CPU 성능 (ms/sec)			
평균	5,851	3,070	**-47.5%**
P95	6,459	3,847	-40.4%
스레드 경합 (%)			
평균	63.03%	21.76%	**-65.5%**
100% 발생	20.09%	0.90%	-95.5%

* 학습 OFF/ON 모두 세 번의 독립 실험 평균값

표 1은 성능 개선을 정량화한 결과로, CPU 시간은 평균 47.5% 감소(5,851ms→3,070ms)하였고 스레드 경합률은 65.5% 감소(63.03%→21.76%)하였으며, 특히 극심한 100% 경합 발생률은 95.5% 감소(20.09%→0.90%)하여 거의 제거되었다.

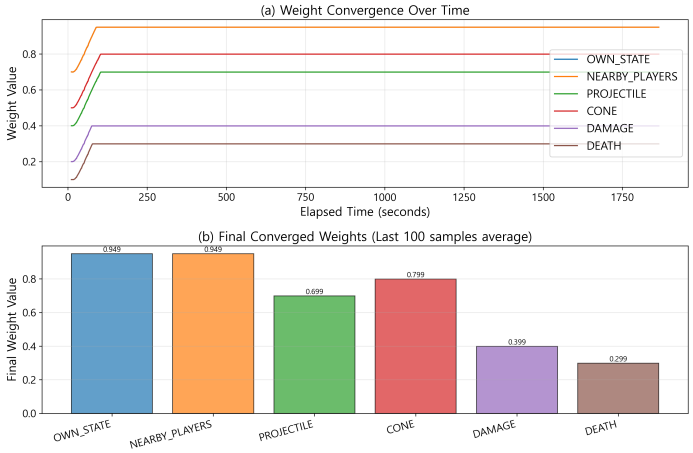

그림 2: 메시지 타입별 가중치 수렴 과정

그림 2는 각 메시지 타입 가중치가 약 5분 이내에 수렴함을 보여준다. 최종 가중치는 OWN_STATE / NEARBY_PLAYERS(0.949), CONE(0.799), PROJECTILE(0.699), DAMAGE(0.399), DEATH(0.299) 순이다. 세 번의 독립 실험에서 학습 OFF는 CV 0.55%, 학습 ON은 3.85%로 높은 일관성을 보였으며, 성능 스파이크(≥5,000ms)는 학습 OFF 1,705건에서 학습 ON 0건으로 완전 제거되었다.

4. 인과관계 분석

AFLL은 학습 오버헤드(38ms/sec, 1.4%)를 추가하면서도 2,781ms/sec(47.5%)의 CPU 절감을 달성했다. 핵심 원리는 순환 인과관계 차단을 통한 비선형 오버헤드 영역 회피이다. OWN_STATE와 NEARBY_PLAYERS는 클라이언트에게 주변 플레이어 정보를 전송하면, 클라이언트가 추가 상호작용 요청을 보내는 피드백 루프를 형성하여 부하를 n^2 복잡도로 증폭시킨다. 부하가 CPU 코어 수를 초과하면 동시성 오버헤드가 비선형적으로 급증하는데, AFLL은 학습된 가중치 기반 타입별 차등 제어로 부하를 임계점(70%) 이하로 유지하여 이를 방지한다. 그 결과 스레드 경합이 63.03%→21.76%(65.5% 감소)로 개선되었으며, 동시성 오버헤드 절감(2,781ms)이 학습 비용(38ms)을 압도하여 순이익을 달성한다.

5. 결론

본 논문은 MMO 게임 서버의 순환 인과관계를 학습하고 제어하는 AFLL 시스템을 제안했다. 1,000명 동시 접속자 실험에서 47.5% CPU 감소, 65.5% 스레드 경합 감소, 성능 스파이크 완전 제거를 달성했으며, 세 번의 독립 실험에서 탁월한 재현성(CV 0.55%와 3.85%)을 보였다. 특히 학습 오버헤드(1.4%)는 동시성 오버헤드 절감(47.5%)에 비해 무시할 수 있는 수준으로, 학습 기반 접근의 성능 부담 문제를 해결했다. AFLL은 사후적 부하 대응을 넘어 사전적 부하 제어 패러다임을 제시하며, 게임 서버뿐 아니라 순환 인과관계가 존재하는 실시간 시스템에 범용 적용 가능하다. 향후 연구로는 다차원 부하 메트릭 학습 확장과 다중 서버 분산 환경 확장이 필요하다.

참고 문헌

[1] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533-536, 1986.

[2] Claypool, M. and Claypool, K., "Latency and Player Actions in Online Games," *Commun. ACM*, vol. 49, no. 11, pp. 40-45, 2006.

[3] Raaen, K., Espeland, H., Stensland, H. K., Petlund, A., Halvorsen, P., and Griwodz, C., "LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition," *Proc. IEEE Int'l Conf. Parallel Processing Workshops (ICPPW)*, pp. 84-93, 2012.

[4] Duarte, E. P., Jr., Pozo, A. T. R., and Beltrani, P., "Smart Reckoning: Reducing the Traffic of Online Multiplayer Games Using Machine Learning for Movement Prediction," *Entertainment Computing*, vol. 33, pp. 100336, 2019.